TURING 图灵程序设计丛书



JavaScript网页动画设计

业界最先进的动画库Velocity.js作者作品
揭秘开发人员如何用动画轻松提升用户体验

【美】Julian Shapiro 著　王沛 译


中国工信出版集团


人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

Julian Shapiro

最先进的动画库Velocity.js作者，资深JavaScript开发人员，曾获Stripe开源奖金。



图灵程序设计丛书

Web Animation using JavaScript: Develop and Design

JavaScript网页动画设计

[美] Julian Shapiro　著

王　沛　译

人民邮电出版社

北　京

图书在版编目（C I P）数据

JavaScript网页动画设计 / (美) 夏皮罗
(Shapiro,J.) 著 ; 王沛译. -- 北京 : 人民邮电出版社,
2016.1
(图灵程序设计丛书)
ISBN 978-7-115-41012-2

Ⅰ. ①J… Ⅱ. ①夏… ②王… Ⅲ. ①JAVA语言－程序
设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2015)第277647号

内 容 提 要

本书由业界最先进的动画库 Velocity.js 的作者所著，书中内容共分为 8 章，简明扼要地总结了在网页上使用动画的技术技巧，让读者掌握如何有效利用动画实现无与伦比的用户体验。具体内容包括：JavaScript 动画优势，Velocity.js 的利用，动画工作流，文本动画，SVG，动画性能。

本书适合所有 Web 开发工程师和动画设计师晋阶学习。

◆ 著　　　　[美] Julian Shapiro
　译　　　　王　沛
　责任编辑　朱　巍
　执行编辑　赵瑞琳
　责任印制　杨林杰

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　　　印刷

◆ 开本：800×1000　1/16
　印张：7.75
　字数：166千字　　　　2016年 1 月第 1 版
　印数：1 – 3 500册　　　2016年 1 月北京第 1 次印刷

著作权合同登记号　图字：01-2015-5819号

定价：39.00元

版 权 声 明

谨以此书献给《反恐精英》的玩家以及喜欢动漫《瑞克和莫蒂》的观众。

致　谢

我衷心感谢Yehonatan Daniv，因为他在Github上为Velocity的用户提供支持与帮助；感谢Anand Sharma，因为他的动效设计作品经常会给我带来启发；还要感谢David DeSandro为本书撰写前言。同时，Mat Vogels、Harrison Shoff、Adam Singer、David Caplan和Murat Ayfer审阅了本书的文稿，在此我也表示由衷的谢意。

序

开发人员首次发现jQuery的`.animate()`函数时是个特别的时刻。我记得自己一发现这个函数就尝试给页面上任何主内容区以外的东西都加上动画效果。我实现了手风琴效果（accordion）、飞出菜单（fly-out menu）、悬停效果（hover effect）、滚动过渡动画（scroll transition）、神奇显示（magical reveal）和视差滚动幻灯片（parallax slider）。这些动画为我原本冷冰冰、静止不动的页面带来了动态的视觉体验，使我感觉作为网页设计师，自己又更上一层楼了。但这些只不过是华而不实的把戏罢了。时至今日，我意识到那时添加的那么多动画并没有真正提高网站的用户体验。

尽管如此，我当时仍是激动不已。究竟是什么让动画如此激动人心？

从我的公寓向外望去，可以看到布鲁克林市中心。我见到下面的街道行人攒动；烟缕从烟囱里向外冒，就好像滚滚巨浪；鸽子振动双翅，飞到窗台上歇息；一台起重机高高吊起建筑物的一部分；一个孤零零的心形气球升向布鲁克林的天空（听上去有些老土，这我知道，不过我确实见到过两次）；汽车在威廉斯堡大桥上穿梭；头上云朵轻轻飘过。

世界在不停地运动。

这也是我们脑中整个宇宙的运作状态。事物在运动。正如窗外这些移动的事物，它们每个都在讲述短短一句话的故事。然而它们合在一起，又在讲述着一个更大的、正在发生的故事。

但是，数字界面却不是这样运作的，它缺少了那些短短的故事。当事情在发生变化时，得自己添补缺漏的情节。比如，点击ATM机上的“下一页”按钮以后，屏幕突然改变了。它成功进入下一页了吗？还是出现错误了？你必须重新读屏并理解上面的信息，才能了解自己的行为带来怎样的结果。这就造成互动与互动之间产生理解的空白。如果使用了动效，就可以消除这种理解的空白，因为动效自然而然地告诉了人们什么东西发生了变化。这就好像在一种状态到另一种状态之间填补上许多小故事一样。

当滑动过渡效果带你来到下一屏时，动画帮助你更好地理解了刚刚发生的事情。正是这种力量使动画格外令人兴奋。就像布局、色彩和字体一样，动画帮助你塑造和引导用户的体验。它不仅仅是让东西运动起来，更需要有效的设计和精心的实现。

不幸的是，在网页动画的历史当中，人们并不总是认为深思熟虑是至关重要的。作为开发人员，我们用过各式各样的方法实现动画，比如Flash、动画GIF、Java小程序、marquee标签，直至最近的CSS、JavaScript和SVG。但是，那些动画充其量只不过是页面上的锦上添花；在最坏的情况下，它们甚至沦为哗众取宠的把戏。动画既要高性能又要人性化，这还是一种相对较新的理念。

因此，有这本书在你面前是一件幸事。Julian Shapiro是网页动画领域最重要的专家之一。在编写与维护Velocity.js期间，他积累了大量一手知识，包括在网站上使用动效的所有窍门和优势。本书不仅会带给你在网页上实现动画的技术诀窍，更重要的是，它蕴含了对动画的深刻见解，只有理解了这些，才能有效地利用动画带来无与伦比的用户体验。

各种动画库和技术已经使动效设计变得空前容易。但并不是每一位开发人员都遵从最佳实践。在过去的几年间，有几种时髦的反模式风行一阵之后又消失无踪。滚动行为被用来实现别样的用途。移动设备上的导航被挤到菜单里面，只有通过手势才能访问。尽管任何偶然知晓`.animate()`函数的开发人员都能轻而易举地添加动画效果，但只有那些能用其提升用户体验的开发人员才可以称为真正具有奉献精神的专业开发人员。本书将助你成为其中一员。

David DeSandro
2015年2月于纽约布鲁克林
David DeSandro是Metafizzy的创始人，也是Mansonry和
Isotope这两款jQuery插件的作者/开发人员

引　言

在网络刚起步时，动画（animation）主要是在开发新手实在没有其他办法时才会使用的，为的是强调页面上的重要部分以吸引人们的注意。即使他们想让动画突破限制而发挥更大作用，也做不到，因为浏览器（以及电脑）的速度太慢，无法流畅地呈现基于网络的动画效果。

我们从闪烁的横幅广告、滚动的跑马灯新闻和Flash介绍视频的旧时光一路走来，取得了长足的发展。时至今日，iOS以及Android中叹为观止的动效设计（motion design）不仅没有降低用户体验，反而使其大大改善。最优秀的网站和应用的开发人员利用动画来提升用户界面的感觉和直觉性。动画在设计开发中的重要性明显提升，这不仅仅是因为硬件的处理能力提高了，更体现了网络开发群体对于最佳实践有了更深的理解。现如今，人们普遍认为最终用户体验的质量比开发网站用什么工具更加重要。当然，尽管这是个看似明显的结论，但事实却并非总是如此。

那么，究竟是什么偏偏让动画变得这么有用？不论是内容块之间的过渡效果、复杂加载次序的设计还是对用户下一步操作的提示，动画都是文字和布局的有效补充，强化了网站的预期行为、彰显了个性、丰富了视觉体验。内容究竟是要以友好的方式弹跳出现呢，还是要猛然甩到屏幕上？这正是动效设计研究的问题。同时，你的选择将会决定应用的总体感觉。

当用户将你的应用推荐给他人的时候，他们经常会试着用“顺滑”或“精致”这样的字眼来形容，但却没有意识到，他们描述的大多是界面上的动效设计。作为外行人，他们没法明确区分应用和应用的动效设计，而这正是优秀的用户界面（UI）设计师孜孜以求的效果：用动画来加强页面所要达到的目标，但同时又不分散用户的注意。

本书为你提供了一些必备的知识。掌握了它，就可以自信地实现动画效果，不仅视觉上效果震撼而且技术上也易于维护。一方面要通过动效设计丰富页面体验，另一方面又要避免累赘的花哨。本书自始至终都努力在这两者之间达到平衡。

为什么所有这一切都如此重要？为什么值得花费时间去优化过渡和淡入淡出效果？以上这

些问题的答案也正是设计师花费几个小时优化字体和颜色的原因：只是因为使产品越来越完美，这种感觉棒极了。是他们让用户啧啧称赞、口口相传："哇，这简直太酷了。"然后马上转头对朋友叫道："你可得看看这个！"

注意　如果不熟悉基本的CSS属性，那就需要先找本介绍HTML和CSS的书看看，然后再来读本书。

目　　录

第1章 JavaScript动画的优势

在本章中，我们将会对比CSS动画和JavaScript动画的优劣，同时介绍JavaScript动画的特点和工作流方面的优势。

简而言之，我们提供所需的背景知识，帮助你理解即将在本书中学到的JavaScript的任何知识。

1.1　JavaScript 动画与 CSS 动画

在网页开发圈子里有一种误解，那就是认为CSS动画是网络中唯一可以实现高性能动画的方法。这种误解使很多开发人员干脆放弃了用JavaScript实现动画，而这会迫使他们做出以下行为。

- 在样式表中管理有关用户界面（UI）互动的所有内容，这样代码很快会变得难以维护。
- 牺牲实时动画的定时控制，因为它只能通过JavaScript实现。（在移动应用中会看到需要响应用户拖拽操作的UI。在为这些UI设计动画时，必须使用定时控制。）
- 放弃基于物理的动效设计，这会使网页上的元素无法表现得像真实世界中的物体一样。
- 不再支持旧浏览器版本，而实际上旧版本浏览器在世界范围内仍大量使用。

事实是，基于JavaScript的动画与基于CSS的动画一样快。之所以人们错误地认为CSS动画在性能上有显著优势，那是因为人们通常拿它与jQuery的动画性能对比，后者确实非常慢。然而，一些彻底绕开jQuery的JavaScript动画库通过与页面的顺畅交互表现出了非凡的性能。

注意　Velocity.js是一个著名的动画库，本书自始至终都在使用它。这是个轻量级的库，但是功能却异常丰富。另外，它与jQuery的动画语法类似，能够大幅降低学习难度。

当然，CSS非常适合实现悬停状态的动画效果（例如：当鼠标位于链接上方时，链接变成蓝色），这也是通常情况下基本的网页所包含的动画。CSS过渡效果可以直接在已有的样式表中实现，这样开发人员就可以避免使用冗余的JavaScript库，使页面不再臃肿。另外，CSS动画不费吹灰之力就可以呈现上佳表现。

但是，本书将会说明为什么JavaScript对于动画来说经常是更好的选择，简单的悬停效果除外。

不要将
JavaScript
与jQuery
混为一谈。

1.2 强大的性能

JavaScript与jQuery常被错误地混为一谈。JavaScript动画很快，是jQuery让它慢了下来。尽管jQuery非常强大，但它并未被设计成高性能的动画引擎。它没有避免“布局颠簸”的机制，这使浏览器在动画处理过程中，过分忙于布局处理的工作。

另外，由于jQuery的代码库除了实现动画以外还有许多其他目的，因此它的内存消耗会在浏览器内触发垃圾回收，导致动画在不可预知的情况下卡壳。最后，由于jQuery团队奉行一个崇高

的追求，那就是帮助新手避免用坏代码毁了他们的UI，因此jQuery放弃了被经常推荐的做法，拒不使用requestAnimationFrame()函数，但另一方面浏览器竞相支持该函数，因为它可以为网络动画大幅提升帧率（frame rate）。

一些彻底绕开jQuery的JavaScript动画库通过与页面的顺畅交互而表现出非凡的性能。其中之一是著名的动画库，也是本书自始至终使用的Velocity.js。这是个轻量级的库，但是功能却异常丰富。另外，它与jQuery的动画语法类似，能够大幅降低学习曲线。

有关性能的问题，我们会在第7章中深入探讨。通过学习浏览器渲染性能的微妙差异，你能够打下坚实的基础，为所有的浏览器和设备创建可靠的动画效果，不论这些浏览器和设备的处理能力如何。

1.3 功能

追求速度当然不是使用JavaScript的唯一理由，它还具有一大堆其他同等重要的功能。让我们大致看几个JavaScript所独有并值得关注的动画功能。

1.3.1 页面滚动

页面滚动是基于JavaScript的动画最为流行的应用之一。最近，网页设计的趋势是创建很长的页面。随着页面向下滚动，让每一部分新的内容自动滚到可视区域中来。

例如Velocity这样的JavaScript动画库提供了一种将元素滚动至可视区域的简单函数：

```
$element.velocity("scroll", 1000);
```

以上代码通过使用Velocity的"scroll"命令让浏览器用1000毫秒的时间滚动至$element的上方边缘位置。注意，Velocity的语法与jQuery的$.animate()函数几乎一模一样，这会在本章后面的部分进行详述。

1.3.2 动画反转

动画反转是用于撤消元素前一个动画的简便方法。通过调用反转命令，你会使元素以动画形式变动回上一个动画开始之前的值。反转的常见用途是动态显示一个模态对话框，然后在用户点击“关闭”后再将其隐藏起来。

如果想实现动画反转，但却没有优化工作流程的话，你会这么做：把上次在每个元素上动态显示的特定属性跟踪记录下来，以备后续反转动画之用。但是在UI代码中跟踪之前的动画状态很快就会变得难以控制。与之形成鲜明对比的是Velocity，通过reverse命令就可以记住一切。

与Velocity中scroll命令的语法相似，reverse命令也是通过将"reverse"作为Velocity的第一个参数传入而调用的：

```
// 第一个动画：设置元素的opacity属性变动至0的动画
$element.velocity({ opacity: 0 });
// 第二个动画：设置元素的opacity属性值变动回初始值1的动画
$element.velocity("reverse");
```

当谈到JavaScript的动画定时控制时，就不仅仅是反转那么简单了：JavaScript还允许对全部正在运行的动画进行全局减速或加速。第4章会介绍这一强大功能的相关知识。

1.3.3　基于物理的动效

动效设计中的物理原则反映了成就优秀用户体验（UX）的核心原则：那就是伴随着用户的输入，界面出现自然的反应。换言之，界面设计遵从物体在真实世界中的运动规律。

在Velocity中有一个简单且强大的入门级物理动效，即基于弹簧运动原理的缓动类型。（我们会在下一章详细探讨缓动的概念。）使用典型的缓动选项，可以传入一个与预先定义的缓动曲线（例如，"ease"或"easeInOutSine"）相对应的字符串。相反，弹簧物理缓动类型接受一个含有两个项的数组作为参数。

```
// 使用500个张力单位和20个摩擦力单位的弹簧物理缓动，设置将元素的宽度变动至"500px"的动画
$element.velocity({ width: "500px" }, { easing: [ 500, 20 ] });
```

缓动数组中的第一项代表弹簧的张力，而第二项代表摩擦力。张力值越大，动画的总体速度就越快，总体反弹幅度就越大。摩擦力值越低，动画尾部振动的速度就越快。通过调整这些数值，可以为页面上每一个动画实现独特的运动效果，这有助于强化不同行为之间的差异。

1.4　易维护的工作流

设计动画是一个不断试验的过程，需要反复修改时间和缓动值才能在页面上达到和谐统一的效果。然而无可避免的是，就当你已经将设计优化到最佳时，客户跳出来要求大改。这种情况下，

代码是否易于维护就变得至关重要了。

基于JavaScript来解决这一工作流难题是非常优雅的，这将会在第4章中详细阐述。在这里先简要解释一下：有技术可以将一个个JavaScript动画链接起来，而且每个动画都有各自不同的持续时间和缓动效果等，这样其中一个动画的定时不会影响其他动画。这就意味着不用重新计算，就可以把某个动画的持续时间改来改去，还能轻易将动画设置为同时进行或是顺次进行。

1.5 小结

当使用CSS设计动画时，肯定要受限于CSS规范所提供的那些功能。但JavaScript是一门编程语言，具有编程语言的天然属性，其第三方程序库可以对动效设计进行无限的逻辑控制。动画引擎利用这一点来提供强大的功能，不仅可以大幅优化工作流，还能拓展交互动效设计的可能性。而这正是本书的目的所在：用尽可能高效的方法设计优美的动画效果。

下一章介绍怎样使用本书选择的JavaScript动画引擎Velocity.js。通过掌握Velocity.js，将会了解如何使用我们前面已经介绍的一些功能以及其他更多内容。

第2章 使用Velocity.js实现动画

本章将介绍Velocity.js所提供的功能、命令和选项。如果已经对基于jQuery的动画比较熟悉，那就已经明白该如何使用Velocity了，它的作用与jQuery的`$.animate()`函数基本一模一样。

但不论是否已具备相关知识，由于本章对Velocity进行了功能分解，你还将会了解到动画引擎行为的细微差异。掌握这些细节会帮助你从新手荣升为专家。即使对JavaScript动画和Velocity都已经非常熟悉，那也浏览一下本章内容，就当帮自己一个小忙。你肯定会有一些意想不到的收获。

2.1 JavaScript 动画库的种类

JavaScript动画库有很多种类。有一些是在浏览器中再现物理交互，有一些使WebGL和Canvas动画更易于维护，有一些专注于SVG动画，还有一些旨在提升UI动画，本书的重点正是这最后一种。

有两个流行的UI动画库，它们是GSAP（请到GreenSock.com下载）和Velocity（请到VelocityJS.org下载）。在使用本书期间，可以免费使用Velocity，因为它采用的是MIT许可证。（GSAP要根据网站的商业形式收取许可费用。）此外，令Velocity引以为傲的是：利用它惊人强大的功能可以写出干净并出色的代码。很多流行网站都采用了这个引擎，例如Tumblr、Gap和Scribd。

噢对了，Velocity就是本书作者编写的！

2.2 安装 jQuery 和 Velocity

可以在jQuery.com下载jQuery，在VelocityJS.org下载Velocity。若要像使用其他任何JavaScript库那样在页面上使用它们，只需要在</body>标签前面添加指向对应库的<script></script>标签。如果想链接至在线版本（而不是位于自己电脑上的本地副本），那么代码可能会像下面这样。

```
<html>
  <head>My Page</head>
  <body>
    My content.
    <script src="//code.jquery.com/jquery-2.1.1.min.js"></script>
    <script src="//cdn.jsdelivr.net/velocity/1.1.0/velocity.min.js"> </script>
  </body>
</html>
```

当同时使用jQuery和Velocity时，将添加jQuery的标签放到Velocity的前面。

就是这样！现在可以开始学习使用Velocity.js实现动画了。

2.3 使用 Velocity：基础知识

为了能让你入门，我们先从基本组成开始讲起：参数、属性、值和链式操作。既然jQuery是这样无处不在，来看一下Velocity与它的关系也挺重要。

2.3.1 Velocity和jQuery

Velocity函数是独立于jQuery的，但两者可以结合使用。通常这么做的好处是可以利用jQuery的链式操作：当你先用jQuery选择了一个元素后，就可以用这个元素去调用.velocity()为它添加动画效果。比如下面的示例。

```
// 将一个变量分配给某个jQuery元素对象
var $div = $("div");
// 使用Velocity设置该元素的动画
$div.velocity({ opacity: 0 });
// 该句语法与jQuery自有的animate函数相同：
// $div.animate({ opacity: 0 });
```

本书中的所有例子都是将Velocity与jQuery结合使用的，因此也都使用该语法。

2.3.2 参数

Velocity接受多个参数。第一个参数是一个对象，用于将CSS属性映射到最终的期望数值上。各个属性以及它们接受的数值类型与在CSS中使用的直接对应。（如果不熟悉基础的CSS属性，请先阅读一本介绍HTML和CSS的书，然后再来阅读本书。）

```
// 设置元素的width属性值变动至"500px"且其opacity属性值变动至1的动画。
$element.velocity({ width: "500px", opacity: 1 });
```

小窍门 在JavaScript中，如果你提供的属性值包含字母（而不是只有整数），那么要将它们用半角引号括起来。

可以将一个指定动画选项的对象作为第二个参数传入。

```
$element.velocity({ width: "500px", opacity: 1 }, { duration: 400, easing: "swing" });
```

还有一种参数简写语法，那就是不将选项对象作为第二个参数传入，而是使用逗号分隔参数语法。这种写法需要列举出动画的持续时间（接受一个整数值）、缓动形式（一个字符串）和动画执行完毕后触发的回调函数（一个函数），以上这些参数可以以任何顺序用半角逗号分隔。（你马上就会学到这些选项都是做什么用的。）

```
// 设置一个持续1000毫秒的动画（且隐式使用"swing"的默认缓动值）
```

```
$element.velocity({ top: 50 }, 1000);
// 设置一个持续1000毫秒且缓动类型为"ease-in-out"的动画
$element.velocity({ top: 50 }, 1000, "ease-in-out");
// 设置一个缓动类型为"ease-out"的动画（且隐式使用默认持续时间400毫秒）
$element.velocity({ top: 50 }, "ease-out");
// 设置一个持续1000毫秒且在动画执行完毕后触发回调函数的动画
$element.velocity({ top: 50 }, 1000, function() { alert("Complete.") });
```

在只需要指定基本选项（持续时间、缓动和完成）时，简写语法是传入动画选项的快速方法。如果除了以上三个动画选项之外，还要传入其他动画选项，那就必须将所有选项切换至对象语法。因此，如果需要指定一个延迟时间，就需要对下面的语法进行更改。

```
$element.velocity({ top: 50 }, 1000, "ease-in-out");
```

更改为下面的样子。

```
// 重新指定以上使用的动画选项，但将延迟时间设置为500毫秒
$element.velocity({ top: 50 }, { duration: 1000, easing: "ease-in-out", delay: 500 });
```

不能这么写成下面的样子。

```
// 错误：一部分动画选项用逗号分隔的语法；一部分用对象语法
$element.velocity({ top: 50 }, 1000, { easing: "ease-in-out", delay: 500 });
```

2.3.3　属性

基于CSS的属性动画与基于JavaScript的属性动画有两点区别。

首先，Velocity针对每个CSS属性，只接受唯一一个数值，这点与CSS不同。

因此，可以这样传入动画选项：

```
$element.velocity({ padding: 10 });
```

或者

```
$element.velocity({ paddingLeft: 10, paddingRight: 10 });
```

但不能按下面这样传入动画选项。

```
// 错误：针对一个CSS属性传入了多个数值。
$element.velocity({ padding: "10 10 10 10" });
```

如果想设置全部四个padding值（top、right、bottom和left）的动画，那么就将它们作为单独的属性列出来。

```
// 正确
$element.velocity({
  paddingTop: 10,
  paddingRight: 10,
  paddingBottom: 10,
  paddingLeft: 10
});
```


CSS还有一些其他常见的多值属性，包括：margin、transform、text-shadow和box-shadow。

在实现动画效果时，将这些复合属性分拆成它们的子属性有助于加强对缓动值的控制。例如，在CSS中，实现父级padding属性内的多个子属性的动画时，仅可以指定一种属性范围的缓动类型。在JavaScript中，可以为每一个子属性设置缓动值，这么做的优势在本章后续讨论CSS的transform属性动画时就变得尤为明显了。

把各自独立的子属性都列举出来也会使动画代码更易读也更易维护。

基于CSS的属性动画与基于JavaScript的属性动画第二个不同点在于：JavaScript的属性名称中，单词之间的连接号去掉了，除第一个单词外，其余单词都首字母大写。例如：padding-left变成了paddingLeft；background-color变成了backgroundColor。另外还要注意，JavaScript的属性名称不能用引号括起来。

```
// 正确
$element.velocity({ paddingLeft: 10 });
// 错误：使用了连字符，而且第二个单词首字母没有大写
$element.velocity({ padding-left: 10 });
// 错误：将JavaScript格式的属性名称用引号括起来了
$element.velocity({ "paddingLeft": 10 });
```

2.3.4　值

Velocity支持px、em、rem、%、deg、vw和vh这些单位。如果没有为数值提供单位，那么就会根据CSS属性类型自动指派一个单位给它。对于大多数属性，px是默认单位。但是，有些属性可

能表示旋转的角度，例如rotateZ，Velocity会自动将deg单位指派给它。

```
$element.velocity({
  top: 50, // 默认使用px单位类型
  left: "50%", // 手动指定了%单位类型
  rotateZ: 25 // 默认使用deg单位类型
});
```

为所有属性值清晰指明单位会让代码更易读，因为当你快速浏览代码时，px单位与其他单位就更易区分。

Velocity胜过CSS的另一优势在于它有四个值运算符+、-、*和/，可以选择其中一个加到属性值前面。这些运算符的作用与JavaScript中的数学运算符一致。也可以将这些值运算符与等号（=）组合使用，来进行相应的数值运算。请参考下面的行间注释。

```
$element.velocity({
  top: "50px", // 无运算符，按预期设置top属性变动至50的动画。
  left: "-50", // 负运算符，按预期设置left属性变动至-50的动画。
  width: "+=5rem", // 将当前的width值转成以rem为单位的，然后再加5个单位。
  height: "-10rem", // 将当前height值转成以rem为单位的，然后再减10个单位。
  paddingLeft: "*=2" // 将当前的paddingLeft值乘以2。
  paddingRight: "/=2" //将当前的paddingLeft值除以2。
});
```

Velocity的简写功能，例如值运算符，可以在动画引擎内完全保留动画逻辑。无需再进行手工值运算，这使代码更加简洁。除此以外，通过告诉Velocity你打算如何实现元素的动画，性能也得到了提升。在Velocity里运行的逻辑越多，Velocity就能越好地优化代码，达到更高的帧率。

2.3.5　链式操作

当一个元素（或一系列元素）链式调用多个Velocity函数时，它们会自动排成队列。这意味着前一个动画一结束，后一个动画马上就开始。

```
$element
  // 设置width和height属性的动画
  .velocity({ width: "100px", height: "100px" })
  // 在运行width和height属性的动画时，设置top属性的动画
  .velocity({ top: "50px" });
```

2.4 使用 Velocity：选项

在本章完成对Velocity的简介之前，让我们大致浏览一下以下几个最常用的选项：duration、easing、begin和complete、loop、delay和display。

2.4.1 duration（持续时间）

可以以毫秒（ms，即千分之一秒）为单位指定duration选项，该选项指定一个动画调用需要花费多长时间才能完成，或者也可以将duration指定为以下三个简写duration之一："slow"（相当于600ms）、"normal"（400ms）或"fast"（200ms）。以毫秒为单位指定duration值时，请提供一个不带单位的整数值。

```
// 设置持续1000毫秒（即1秒）的动画
$element.velocity({ opacity: 1 }, { duration: 1000 });
```

或

```
$element.velocity({ opacity: 1}, { duration: "slow" });
```

使用意义明确的duration简写形式有个好处，那就是当你重新查看代码时，它能清楚地表明动画的节奏究竟是慢还是快？如果只使用这几个简写形式，那么网站上的动效设计就自然而然地形成了统一的节奏，因为每个动画都肯定是这三种速度之一，而不是什么其他随意设定的数值。

2.4.2 easing（缓动）

easing选项都是数学函数，它们定义了在动画的整个持续时间中的不同时间段内动画应该执行得有多快或多慢。例如，"ease-in-out"缓动类型就表示动画一开始要逐渐加速（ease in），最后要逐渐减速（ease out）。相比之下，"ease-in"缓动类型则使动画在一开始加速到目标速度，然后维持这一速度直至动画结束。"ease-out"缓动类型恰恰与之相反：动画以恒定速度开始并持续一段时间，然后在动画结束之前逐渐减速。

就像在第1章中讨论到的基于物理的动效一样，这些缓动也赋予你力量，使你能将个性注入到动画当中。举例来讲，使用线性运动作为缓动类型的动画看上去是多么机械啊。（线性缓动所产生的动画在动画开始、运行以及结束时，速度一致。）之所以会产生机械的感受是因为人们马上联想到现实世界中的线性运动：自制导的机械物体通常是在直线上以同样的速度移动，这是因

为它们既不用于审美也不是有机体，因此无法自行变速运动。

与之相反，只要是生物，不论是人体还是风中摇曳的树枝，在现实世界中都绝不会以恒定速度运动。摩擦力以及其他外力使它们的运动速度一直在变。

优秀的动效设计师尊崇的是有机的运动，因为它会让人感觉界面是随着用户的互动而自然地作出反应。例如，在移动应用中，将菜单划离屏幕时，你预计它会迅速加速远离你的手指。如果菜单栏没有这样反应，而是就好像机械手臂一样，以恒定速度离开你的手指，那么你会感觉划屏只不过是触发了一系列运动，但那些运动是在你掌控以外的。

第3章将介绍更多有关缓动类型的强大功能。现在，让我们先大致浏览一下Velocity的所有可用的缓动值。

- jQuery UI中的三角函数缓动（trigonometric easing）。要想得到所有这些缓动方程的列表以及它们变速运动的互动演示，请参考easings.net上的示例。

```
$element.velocity({ width: "100px" }, "easeInOutSine");
```

- CSS缓动："ease-in"、"ease-out"、"ease-in-out"和"ease"（"ease-in-out"的另一个更缓和的版本）。

```
$element.velocity({ width: "100px" }, "ease-in-out");
```

- CSS的贝赛尔曲线（Bézier curves）：通过贝赛尔曲线缓动，可以完全控制一个缓动加速曲线的结构。一条贝塞尔曲线可以通过指定图中四个等距点的高度来确定，Velocity所接受的参数格式是一个含有四个小数的数组。请访问cubic-bezier.com获取创建贝赛尔曲线的互动指南。

```
$element.velocity({ width: "100px" }, [ 0.17, 0.67, 0.83, 0.67 ]);
```

- 弹簧物理（Spring physics）：这种缓动类型模仿的是弹簧在拉伸以后被突然释放的弹动行为。就像是经典的定义弹簧运动的物理方程一样，这种缓动类型允许你提供一个含有两个值的数组作为参数，形式为[张力、摩擦力]。张力越大（默认：500），整体速度和弹动幅度就越大。摩擦力越小（默认：20），弹簧结尾处的震动速度就越快。

```
$element.velocity({ width: "100px" }, [ 250, 15 ]);
```

- 如果不想试验各种张力和摩擦力数值，"spring"缓动就是一种随手可用的弹簧物理缓动的预设。

```
$element.velocity({ width: "100px" }, "spring");
```

请记住，也可以将缓动选项作为一个在另一个选项对象参数中显式定义的属性传入，比如下面的代码。


```
$element.velocity({ width: 50 }, { easing: "spring" });
```

不要见到这么多缓动类型就感到不知所措。大多数时候，只会用到CSS的缓动类型和"spring"缓动，因为它们适用于绝大多数动画使用场景。最复杂的缓动类型，也就是贝赛尔曲线，通常是被那些脑中已有明确缓动形式的开发人员使用，而且这些人还得不嫌麻烦、不怕折腾。

注意　必须将本节中提到的其他Velocity选项明确地传入到某个选项对象中。与已经介绍过的那些选项不同，其他这些选项不能以简写的逗号分隔语法提供给Velocity。

2.4.3　begin（开始）和complete（完成）

使用begin和complete选项可以指定要在动画中的特定时间点触发的函数：为begin设置的函数会在动画开始之前触发。与之相反，为complete设置的函数会在动画完成时调用。

使用这两个选项，每次调用动画时都会调用一次指定函数，即使同时制作多个元素的动画也是如此。

```
var $divs = $("div");
$divs.velocity(
  { opacity: 0 },
  // 在动画开始之前打开一个警告框
  {
begin: function () { console.log("Begin!"); },
     // 动画一结束就打开一个警告框
     complete: function () { console.log("Complete!"); }
  }
  );
```

回调函数

这些选项通常被称为“回调函数”（或“回调”），因为它们是在未来某个特定事件发生时被“调用”的。当需要依据元素的可见性来触发事件时，回调函数就能派上用场。例如，如果一个元素最初不可见，然后制作opacity变动至1的动画，那么可能就适合在随后触发一个UI事件，使用户一看见这个元素就能对该新内容进行修改。

请记住，如果想让多个动画一个个排成队列，那么就不需要使用回调函数，因为如果多个动画指派给了单独一个或一组元素，那么它们会自动按顺序触发。回调函数不是用来给动画排队的。

2.4.4 loop（循环）

将loop选项设置为一个整数，该整数是几，动画就应该在调用的属性映射中的值与调用之前元素的值之间交替几次。

```
$element.velocity({ height: "10em" }, { loop: 2 });
```

如果元素的初始height值为5em，那么它的高度就会在5em和10em之间交替两次。

如果将begin或complete选项与某个循环调用一起使用，那么它们每个只会被触发一次，分别是在整个循环序列最开始时和结束时；它们不会在每次循环交替时被重新触发。

除了可以传入整数以外，还可以将布尔值true传给loop，这样就会触发无限循环。

```
$element.velocity({ height: "10em" }, { loop: true });
```

无限循环会忽略complete回调，因为循环不会自然停止。不过可以通过Velocity的stop命令，手动停止循环。

```
$element.velocity("stop");
```

有限循环对于需要重复链式动画代码的动画序列很有用。例如，如果想让一个元素上下弹跳两次（也许是想提示用户有新消息等待阅读），未经优化的代码会像下面这个样子。

```
$element
  // 假设 translateY 的初始值是 "0px"。
  .velocity({ translateY: "100px" })
  .velocity({ translateY: "0px" })
```

```
  .velocity({ translateY: "100px" })
  .velocity({ translateY: "0px" });
```

更简洁方便且易于维护的代码版本则是下面这个样子。

```
// 重复（循环）该动画两次
$element.velocity({ translateY: "100px" }, { loop: 2 });
```


有了这个优化的版本，当你改变主意想修改最大变动值（目前是"100px"）时，就只需在代码的一个地方更改一个数值即可。如果在代码中有多处重复的动画，那么循环为你的工作流带来的好处就立马显现出来了。

无限循环对于加载指示器（loading indicator）大有帮助，因为通常情况下，只要数据还没有加载完毕，就会一直显示加载指示器的动画。

首先，通过让元素的不透明度在可见与不可见之间无限循环，使加载元素看上去像是在做有规律的跳动。

```
// 假设 opacity 的初始值是1（完全可见）
$element.velocity({ opacity: 0 }, { loop: true });
```

稍后，只要数据完成加载，就可以立即停止动画，然后隐藏该元素。

```
$element
  // 首先停止无限循环……
  .velocity("stop")
  // …… 这样可以为元素添加一个新动画，
  // 在该动画中，您可以使该元素变动回不可见
  .velocity({ opacity: 0 });
```

2.4.5　delay（延迟）

将delay指定为多少毫秒，在动画开始之前就会暂停多长时间。delay选项的目的是将动画的定时逻辑完全保留在Velocity内，而不是在Velocity的动画开始时依赖jQuery的$.delay()函数来更改。

```
// 等待100毫秒后再设置opacity的值变动至0的动画
$element.velocity({ opacity: 0 }, { delay: 100 });
```

可以同时设置delay和loop选项，这样可以在循环交替之间创建一个停顿。

```
// 循环四次，每次循环之间都等待100毫秒
$element.velocity({ height: "+=50px" }, { loop: 4, delay: 100 });
```

2.4.6 display（显示）和visibility（可见性）

Velocity中的display和visibility选项与CSS中的同名属性一一对应，接受的值也相同，包括："none"、"inline"、"inline-block"、"block"、"flex"等。另外，Velocity还允许将值设为"auto"，从而告诉Velocity将display属性的值设为与元素的默认值一致。（参考：anchor和span的默认值为"inline"；而div和大多数其他元素的默认值为"block"。）Velocity的visibility选项也像CSS中的同名属性一样，接受"hidden"、"visible"和"collapse"等值。

在Velocity中，如果将display选项设置为"none"（或者如果将visibility选项设置为"hidden"），那么当动画完成时，元素的CSS属性也会进行相应设置。这种做法有效地实现了动画一完成就将元素隐藏；同时，要将元素的opacity值变动至0时（即想要让元素在页面上淡出），搭配这种做法也很实用。

```
// 使元素淡出直至opacity变为0，然后将该元素从页面的文档流中移除
$element.velocity({ opacity: 0 }, { display: "none" });
```

注意 上面的代码可以有效替代下面这段jQuery代码。

```
$element
  .animate({ opacity: 0 })
  .hide();
```

快速回顾visibility和display

CSS中的display属性决定了一个元素如何影响其周围元素及其包含的元素。相比之下，CSS中的visibility属性则仅影响元素是否可见。如果元素被设置为"visibility: hidden"，那么它还是会占据页面上的位置，只不过这个位置上显示为空白，该元素不会被人们看到而已。但是，如果元素被设置为"display: none"，则会在文档流中移除该元素，所有该元素内部以及周围的元素会填充到移除元素的位置，就好像该元素从来没有存在过。

请注意，如果不想将元素从文档流中移除，可以简单地通过将其visibility设置为"hidden"使其隐藏并无法交互。在想要隐藏元素但想让它继续占据页面位置时，这种做法就很有用。

```
// 使元素淡出直至opacity变为0，然后使它不再能够交互
$element.velocity({ opacity: 0 }, { visibility: "hidden" });
```

现在，让我们以相反的方向考虑动画（不再隐藏元素而是要显示元素）：当display和visibility的值分别被设为"none"或"hidden"以外的值时，该值是在动画开始**之前**设置的，因此元素在随后整个动画期间都是可见的。换言之，当元素在之前通过移出可视区域的方式隐藏时，可以用这种方法将隐藏取消。

下面的例子中，在元素开始淡入前，其display被设置为"block"。

```
$element.velocity({ opacity: 1 }, { display: "block" });
```

上面的代码可以有效替代下面的jQuery代码。

```
$element
  .show()
  .animate({ opacity: 0 });
```

小窍门　要查看Velocity动画选项的完整概述，请参考VelocityJS.org上的文档。

包含动画逻辑

就像Velocity的delay选项一样，Velocity将CSS的display和visibility设置也包含进来，使动画逻辑得以全部在Velocity中实现。在产品代码中，不论元素是淡入还是淡出，基本上总是伴随着display或visibility属性的变化。利用Velocity的这些简写形式能够帮助保持代码简洁并易于维护，因为它不太依赖于外部的jQuery函数，而且也摆脱了那些经常使动画逻辑冗余不堪的重复性的辅助函数（helper function）。

注意，针对上面示例中的不透明度切换的动画，Velocity也有简写形式。它们的作用与jQuery中的fadeIn和fadeOut函数相同。只需要将对应的命令作为第一个参数传入Velocity，然后在需要时传入一个选项对象即可。

```
$element.velocity("fadeIn", { duration: 1000 });
$element.velocity("fadeOut", { duration: 1000 });
```

2.5 使用 Velocity：其他功能

Velocity.js的其他值得一提的功能包括：reverse（反转）命令、scrolling（滚动）、color（颜色）和transform（变换，包括translation“平移”、rotate“旋转”和scale“缩放”）。

2.5.1 reverse（反转）命令

要让元素变动回到上次调用Velocity之前的值，请将"reverse"作为第一个参数传入Velocity中。"reverse"命令的作用与标准的Velocity调用相同；它可以取用选项，并与其他链式Velocity调用一起形成队列。

反转命令默认采用在元素的上次Velocity调用中使用的选项（duration、easing等）。但是，也可以传入新的选项对象来覆盖掉原有的这些选项。

```
// 使用上次Velocity调用的选项设置变动回原始值的动画
$element.velocity("reverse");
```

或者

```
// 与上面代码的作用相同，只是用值2000毫秒替换了先前调用的duration。
$element.velocity("reverse", { duration: 2000 });
```

注意 一旦使用了reverse命令，那么之前调用中的begin和complete就会被忽略；reverse永远不会重新调用回调函数。

2.5.2 scrolling（滚动）

要让浏览器滚动至一个元素的上缘位置，请将"scroll"作为Velocity的第一个参数传入。"scroll"命令的行为与标准Velocity调用的行为相同；它可以取用选项，并与其他链式Velocity调用一起形成队列。

```
$element
  .velocity("scroll", { duration: 1000, easing: "spring" })
  .velocity({ opacity: 1 });
```

上面的代码会让浏览器移动到指定元素的上缘位置，持续时间为1000毫秒，缓动类型为"spring"。然后，一旦元素移动到可视区域中，它就会完全淡入。

要想在一个含有滚动条的父元素内滚动至某个元素，可以使用container（容器）选项，它接受一个jQuery对象或原生对象。注意，在容器元素的CSS中，必须将属性设置为relative、absolute或fixed；如果设置为static则不起作用。

```
// 将$element元素滚动至$("#container")的视图中
$element.velocity("scroll", { container: $("#container") });
```

无论是相对于浏览器窗口滚动还是相对于父元素滚动，scroll命令在这两种情况下总是由要被滚动到可视区域中的元素来调用。

默认情况下，滚动是在*y*轴上发生的。传入axis: "x"选项则会在横向而非纵向上滚动。

```
// 滚动浏览器至目标div的左边缘。
$element.velocity("scroll", { axis: "x" });
```

最后，scroll命令还采用了一个独有的offset选项（单位为像素），该选项的作用是使目标滚动位置发生偏移。

```
//  滚动至元素上边缘以上50像素处。
$element.velocity("scroll", { duration: 1000, offset: "-50px" });
// 滚动至元素上边缘以下250像素处。
$element.velocity("scroll", { duration: 1000, offset: "250px" });
```

2.5.3　color（颜色）

Velocity支持以下这些CSS属性的颜色动画：color、backgroundColor、borderColor和outlineColor。在Velocity中，颜色属性只接受表示颜色的十六进制字符串，例如，#000000（黑色）或#e2e2e2（浅灰）。要更加精细地控制颜色，可以单独控制一个颜色中的红、绿和蓝分量，还可以控制alpha分量。红、绿、蓝分量的值的范围是0至255；alpha（相当于不透明度）值的范围是从0到1。

请参考下面例子中的行内注释。

```
$element.velocity({
  // 设置backgroundColor变动至黑色的十六进制值的动画
  backgroundColor: "#000000",
  // 同时设置将背景的alpha（不透明度）值变动至50%的动画
  backgroundColorAlpha: 0.5,
  // 另外设置将元素的文字颜色中的红色分量值变动至总值一半的动画
  colorRed: 125
});
```

2.5.4　transform（变换）

CSS中的transform属性可以控制元素在2D和3D空间中进行平移、缩放和旋转。它包括很多子属性分量，其中Velocity支持以下这些属性分量。

- translateX：沿*x*轴移动元素。
- translateY：沿*y*轴移动元素。
- rotateZ：围绕*z*轴旋转元素（实际上就是在2D平面上顺时针或逆时针旋转）。
- rotateX：围绕*x*轴旋转元素（实际上就是在3D空间中靠近或远离用户旋转）。
- rotateY：围绕*y*轴旋转元素（实际上就是在3D空间中向左或向右旋转）。
- scaleX：使元素的宽度尺寸成倍增加。
- scaleY：使元素的高度尺寸成倍增加。

在Velocity中，可以将这些分量作为某个属性对象内的单独属性设置其动画。

```
$element.velocity({
  translateZ: "200px",
  rotateZ: "45deg"
});
```

2.6　使用 Velocity：不用 jQuery（中级技巧）

如果你是一位中级开发人员，喜欢在没有jQuery的帮助下使用JavaScript，那么你会感到开心，因为Velocity也可以在页面中没有jQuery的情况下使用。相应地，正如本章前面例子中所展示的那样，此时动画的链式操作不能再由jQuery元素来调用，而是要直接将目标元素（一个或多个）作为第一个参数传入动画调用中。

```
Velocity(element, { opacity: 0.5 }, 1000); // Velocity
```

即使在不用jQuery的情况下，Velocity依然保留了与jQuery中$.animate()函数一样的语法；区别在于所有的参数都向右移了一个位置，留出了第一个位置来传目标元素。此外，动画是由全局性的Velocity对象来触发的，而不再由特定的jQuery元素对象触发。

在没有jQuery的情况下使用Velocity时，不需要再设置jQuery元素对象的动画，而是设置原生文档对象模型（DOM）元素的动画。可以使用以下函数检索原生DOM元素。

- document.getElementByID()：按ID属性检索元素。
- document.getElementsByTagName()：检索具有特定标签名称（如a、div、p）的所有元素。
- document.getElementsByClassName()：检索具有特定CSS类的所有元素。
- document.querySelectorAll()：该函数与jQuery的选择器引擎的作用基本相同。

让我们继续探索一下document.querySelectorAll()，因为它可能是在没有jQuery帮助的情况下，选择元素的有力武器。（该函数性能高效，被各种浏览器广泛支持。）就像jQuery元素选择器的语法一样，只需要将一个CSS选择器（即在样式表中用来选择目标元素的选择器）传入querySelectorAll，然后所有匹配的元素就会以一个数组的形式返回。

```
document.querySelectorAll("body"); // 获取body元素
document.querySelectorAll(".squares"); // 获取所有"square"类的元素
document.querySelectorAll("div"); // 获取所有div
document.querySelectorAll("#main"); // 获取id为"main"的元素
document.querySelectorAll("#main div"); // 获取"main"中所有的div
```

如果将其中一个查询的结果分配给某个变量，那么就可以重新使用这个变量来设置目标元素的动画了。

```
// 获取所有div元素
var divs = document.querySelectorAll("div");
// 设置所有div的动画
Velocity(divs, { opacity: 0 }, 1000);
```

既然不能再使用jQuery元素对象，那么你可能想知道怎样才能把一个个动画像下面这样链起来。

```
// 这些动画一个链一个
$element
  .velocity({ opacity: 0.5 }, 1000)
  .velocity({ opacity: 1 }, 1000);
```

要想在没有jQuery的情况下重新实现这种操作，只需要一个接一个地调用动画即可。

```
// 同一元素上的动画会自动链在一起
Velocity(element, { opacity: 0 }, 1000);
Velocity(element, { opacity: 1 }, 1000);
```

2.7　小结

现在你已经了解了使用JavaScript开发网页动画的优势，还掌握了一些Velocity的基本知识，你已经有了充分准备，可以继续探索专业动效设计那令人着迷的理论基础了。

第3章
动效设计理论

实用和优雅是每一位优秀动效设计师追求的目标，而本章探索了实现这些目标的几个技巧。由于此处的焦点在于动效设计理论而非实施，因此并没有代码实例。不论使用的是何种语言、设备或平台，都可以宽泛地概括出这些技巧。

3.1 动效设计提升用户体验

让我们先来研究动效设计这个词组：设计动效就是决定一个对象的哪些视觉属性应该发生改变，应该以何种速度发生改变。例如，想要通过改变颜色让用户注意到一个按钮，你也许会改变其背景颜色，在1000毫秒的时间里从红色变成蓝色，使用`ease-in-out`（先加速后减速）的缓动类型。在此例中，`background-color`就是目标属性；红色就是理想的结束值；该属性向结束值过渡的持续时间为1000毫秒，由`ease-in-out`缓动类型决定了其加速度曲线。优秀的动效设计师选择其中每一项都经过了深思熟虑，这不是因为它们看起来漂亮或者符合流行趋势，而是因为它们强化了UI的意图。与之相反的是那些拍脑袋做出的动效设计，不仅不伦不类而且也不雅观，会分散用户注意力。

讨论UI设计的教程成百上千，但讨论动效设计的却凤毛麟角。这也并不奇怪，因为对于网页而言，动效设计没有UI设计那么重要。直到最近，浏览器和设备的速度才足够快，足以撑起丰富的动效设计。但是在UI设计为页面互动架起结构化基础的同时，动效设计也丰富了这一基础，它通过装备和装饰页面使其更加易用和舒适。其中“装备”就是动效设计提供的实用性，“装饰”就是优雅美观。

> 要么实用，
> 要么优雅，
> 要么死路
> 一条。

优秀的移动应用不仅实用而且优雅，使用户感觉像是在与一个有生命、会呼吸、触手可及的界面进行交互。如果界面对用户的反应就像真实世界中那样，那么用户参与互动就会更加投入。与之相反，没有任何动效设计的页面总是提醒着用户，她只是在将鼠标拖过屏幕或者只是在一片玻璃上敲着手指头。没有动效设计的UI，只会让用户痛苦而无法入戏。

动效设计的实用性利用的是用户心理。当用户按下一个按钮时，她是否确信这个按下的动作已经被UI认出了？有一个简单的方法使她放心，那就是设置一个按钮过渡到按下状态的动画。当用户等待内容加载时，她是明确知道进度仍在继续还是有一种应用已经僵住的不详感觉？动效设计可以提供当下UI状态的视觉提示，从而作用于人们的心理预期。

优雅的动效设计则是锦上添花，它能够让应用从仅是看上去很好升格为感觉上很好。这正是使人情不自禁发出赞叹的地方，因为它让用户意识到技术拥有多么强大的魔力。

让我们同时掌控实用与优雅。同我一起继续深入了解吧。

3.2 实用

怎样才能确定添加的动效设计对网站而言是有价值的？这里有几个技巧。

3.2.1 借鉴惯例

让你最喜爱的网站或应用上的动效设计为你提供灵感。流行的动效设计惯例是值得一用的，原因在于它们在用户脑中已经形成固定含义。这些惯例一次又一次暴露在用户面前，使他们对某些特定动画“应当”怎样形成了预期。如果你使用惯例做法的目的与用户原本预期的不同，人们就会感觉你的应用不够直观。

从别处借鉴的动效设计效果越多，你的应用给人的感觉就越熟悉。而人们对应用越熟悉，就能越快地适应它并对使用它充满自信。尽管确实有些实用工具很新颖，但日常UI元素的动效设计不应该是新颖的。而如果动画本身含义很浅显，比如页面加载顺序动画，或者不太会产生误解，比如状态指示器动画，那么请保留动画的新颖性。

3.2.2 预览结果

当页面上的某个元素看上去目的不太明确时，请为用户提供交互结果的预览。这样做可以确保元素像用户所认为的那样行动。一个简单的例子就是文件传输按钮，当鼠标悬浮其上时会发出

像是无线电波脉冲的视觉效果。这种做法利用了常见的图形设计喻义，告诉用户点击按钮后数据传输就会开始。

另一种不那么隐晦的预览方式是显示用户采取行动之后会发生的部分动画效果。例如，如果用户点击按钮，表示正在进行的文件传输指示器动画就开始运行，那么可以采用这样的动效设计，当鼠标悬停在触发元素上面时就部分运行传输进行时的动画。当用户将鼠标从元素上移开时，就反转刚刚发生的部分动画，使文件传输指示器回到默认状态。这种预览技巧帮助用户马上了解她的行为将要触发的结果，同时也帮助她确认了UI元素的目的。用户越是感到自信，就越有掌控感；而她越有掌控感，体验就越愉悦。

3.2.3　无聊时的消遣

当用户在页面上进行机械性的、毫无参与感的操作时（比如在填写长长的表单时），可以使用颜色和动画让她提提神，增添点兴趣。例如，可以在她成功填写每个表单字段后，设置一个复选标记的动画。这会让用户头脑与界面保持表面上的互动，减少完成当前任务的无聊感。同理，当用户等待内容加载时，可以显示一个吸引眼球的加载指示器。一个绝佳的例子来自名为Lyft的流行拼车应用。当该应用加载至内存时，在空白画布上会有个气球有节奏地上下浮动。

让用户的头脑放松并感受这种重复运动带来的愉悦，这可以使她对你的内容更感兴趣。不管这看上去多么肤浅，它确实奏效。但要意识到，这种技巧只能用于用户将无可避免地感到一段乏味的情况。不要随意把它当创可贴用，每当想给UI加点料时就贴上一贴，只会坏事。

让我们考虑另外一个例子：当Facebook往动态消息里加载文字内容时，它先让一段虚拟文字时而模糊时而清晰地反复运动，直至真正的文字显示出来。这种有节奏的模糊动画不仅表明界面正在努力工作（而不是卡住了），而且还明确告诉用户她在等待的是UI的哪一部分内容。这种技巧被称为行内状态指示。请将这种技巧与无处不在的单状态指示器进行比较，后者在网页一出现时就存在了：它就是一个单一的、不断循环的动态图片，加在一个空空如也、令人不适的空白页面上。用户已经对此厌倦了。与之相反，行内状态指示让你尽可能多地显示界面内容，只把等待内容加载的特定区块屏蔽起来即可。这种做法不仅更细致，也提供给用户更多内容，让她在转着大拇指等待页面全部加载时，有内容可看。

这里的诀窍很简单：给用户参与的内容越多，他们就越不容易感到厌倦。

3.2.4 用本能反应

人脑有一个区域专门用于视觉处理。不论我们愿意不愿意，在面对突然运动时都会作出反应，这是我们的本能。所以，如果页面上有个重要行为需要用户马上关注，可以考虑使用运动来提请注意。有一种提醒用户的常见做法，即通过重复地上下移动，让元素好像在“跳动”一样。这种做法与设置元素颜色的动画截然不同，因为后者并没有作用于人的本能，我们生存的内在机制并不认为颜色的改变值得我们本能地立即予以关注。（然而，在许多国家中，由于特定颜色反复出现，人们被训练得见到红色就理解为“停止”，见到绿色就理解为“前进”。这说明社会活动可以强化某种意义。有鉴于此，当我们在设计动效时，可以将这一点考虑在内。）

再深入挖掘一点儿人的心理，用户将*靠近*自己的运动理解为需要马上行动的紧急提示，而将*远离*自己的运动理解为与自己无关，因此也就不必要采取行动。

3.2.5 使人对互动充满欲望

如果按钮有丰富的渐变色彩而且又大又软，这就使用户想去按它。像这样的元素在点击时，会产生愉悦感，一方面是按压带来的满足，另一方面是绚丽的色彩赏心悦目。在此例中可以学到一个刺激用户互动的诱因：点击按钮越吸引人，用户就越会这样做。充分利用这一现象，在想让用户采取重要行动时发挥作用，比如注册新账户的按钮，或者对购物篮里的货物结账的按钮。

3.2.6 体现重要性

如果用户行为会带来无法逆转的结果，请使用感觉上同等重要的动效设计来强调这一点。例如，点击删除按钮的动画要比鼠标悬停在普通导航下拉列表上的动画给人更重要的感觉。后者可能只需要简单的颜色变化，但前者也许就要包含按钮尺寸突然放大和元素边框加粗的双重效果。根据互动行为的重要程度区分动效设计，这会帮助用户直观地了解可采取行动的重要性。这一技巧，连同本章详述的其他技巧，都旨在提升用户的理解与自信。

3.2.7 减少同时发生的动画

从某种意义上说，用户总是在试图理解你的UI。不论是有意识还是无意识，他们会认为你所选择的每一个设计和动画都有其意图。因此，如果给用户看了大量动画，其中许多元素同时在眼前运动，这其实会削弱她对所有这些运动的意义的理解。

简而言之，如果使用动效设计来强调某件事情的重要性，务必不要一次强调太多不同的东西。

如果确实有很多需要强调，考虑让动画一步一步地显示或者减少动画的总数量。

3.2.8 减少动画种类

上面讲到减少同时发生动画的数量这一最佳实践做法，与此相关的还有减少动画种类：使用的动画种类越少，就越能确保用户充分理解UI上每个动画的含义。例如，如果使用了一种动画将大幅图片呈现在眼前，但使用了另一种动画来呈现小图片，那么请考虑将两种合二为一。如果原本区分动画仅是为了审美考虑而没有提升实用性，那么去掉一种就会让你成功降低UI上不必要的复杂性，而且在这个过程中也强化了行为的一致性。一致性带来模式识别和理解，理解又能增强用户的自信心。

3.2.9 镜像动画

与限制动画种类有那么点儿关系的还有选择动画属性和选项组合的一致性。例如，如果有个模态窗口是通过过渡opacity和scale这两个属性显示出来的，那么请确保在模态窗口隐藏时，也是通过将这两个属性恢复到初始值来实现。这两者就好像是同一硬币的两面，其动效的属性不应有区别。因为一旦动效的属性发生了改变，用户就会有疑问，究竟是什么导致了这种不同？而产生这种不必要的疑问正说明用户体验欠佳。

处理与平移相关的属性时（例如，CSS中的translateX、left和marginLeft），必须要一丝不苟地执行镜像动画：如果模态窗口通过从页面顶部下滑的运动方式进入视图，那么它离开视图就应通过向上滑回顶部的运动方式实现。反之，如果通过继续向下滑出页面让模态窗口离开视图的话，你对用户传达的意思是它被送到了一个*新*去处，而非回到它原本的老地方。通常情况下，如果用户完成了某种操作，比如更改了账户设置，你想表达的就是模态窗口回到它之前来的地方。然而，如果用户是发送邮件，那么让模态窗口下滑出页面就适合该场景，因为它强化了这样一个意思：邮件从原来的地方（用户）被送到了*新*地方（收件人）。

3.2.10 限制持续时间

设计师经常犯这样一个错误，就是让动画的持续时间过长，从而造成用户不必要的等待。永远不要让UI上花里胡哨的东西拖慢页面的显示速度。如果在一串动画序列中，有许多内容要淡入视图，那么请确保整个动画序列*总共*的持续时间很短。

同理，如果UI中有一部分（比如头像）由于用户与页面的交互方式，需要频繁地渐变进入或

离开视图，那么就要格外小心，不要让动画的持续时间过长。第一次看到一段动效设计感觉固然很好，但如果用户每次与移动应用交互时都要看上好几遍，那很快就会让人厌烦了，尤其是当用户要反复等待动画播放完毕的情况，因为他们感觉这显著拉长了整个UI的等待时间。

测试时要看动画播放几十次，之后要确定动画持续时间是否合适是比较困难的。有鉴于此，有一条很好的经验法则可以借鉴，那就是在正式发布网站之前，将所有动画的速度加快25%。这会确保动画总是倾向于快一点播放。（请参见第4章中的小窍门：如何快速让动画进行时间平移。）

3.2.11 限制动画

如果将某个动画整个移除不会影响用户理解界面，那么请考虑将它去掉，用样式的瞬时变化来替代。整个UI上的动画越多，用户就对它们越习以为常。用户越习以为常，就越不会注意它们，进而就不太可能区别不同动效设计类型的差异以及每种类型所代表的意义。

动效设计中的绝大多数动效都应该是很微妙的，例如，鼠标悬停时颜色仅发生轻微变化，因此仅存的极少数大手笔动效设计就应当夺人眼球，传达它们预定的信息。

不要
华而不实。

3.3　优雅

华而不实与富有意义的动效设计，两者间的分界线很容易分辨，即：看这段动效设计是否符合第3.2节中谈到的最佳实践之一？如果不符合，请把它去掉，因为它华而不实，有损UI的易用性。

3.3.1　不要华而不实

要锻炼识别华而不实的能力，请把最流行的应用下载下来，每一个都持续彻底地玩一玩，然后判断它们的动画特色比自己应用中的更鲜明还是更不鲜明。密切关注每个动画传达的意义及其理由。如果你感觉这些应用中使用动画的程度远远低于自己的应用，那么就要考虑将你的UI中的动效设计调整得更缓和一些。

但是，“不要华而不实”这一金科玉律也有个例外情况，欲知详情，请继续往下读！

3.3.2　唯一华而不实的机会

页面加载，即所有元素从初始的不可见状态直至以动画方式进入视图中这段时间，是你唯一一次可以使用出格且华而不实的动画的时机。为什么？因为这样的加载只发生一次，不会在用户与网站交互时反复出现、碍手碍脚。同时，这也是你发挥动效设计技巧，使网站给人留下深刻第一印象的时机。

如果针对内容如何以动画方式进入视图，你已经有了绝佳的点子，那就用在这里。但是，务必确保遵守本章中所有其他规则，尤其是限制持续时间这一条。

3.3.3　考虑个性化

如果要设计一个公司网站，就不能用反弹效果让元素跳着出现，因为反弹效果很顽皮，而这通常不是一个公司想要传递的品质。如果要设计一个教育或政府应用，就不能用那种开始很快、结尾很慢的缓动效果（这些效果给人一种于面前疾驰而过的、充满未来感的光滑感），可能对于眼前的内容而言，这些效果太炫太时髦了。

在选择动画时，要时刻考虑它们所表达的个性化特点。作为设计师，你很难判断自己作品的格调，因此请第三方尽早且时常地给予反馈是个不错的主意。询问测试用户你的UI是否给人以恰当的专业感、亲切感或时尚感，根据你对其中每种特质的偏好，调整动效设计。

3.3.4 不要拘泥于不透明度动画

让元素过渡进入视图的最常用方式是设置opacity属性从0变动至1的动画，而让元素过渡离开视图的最常用方式是设置opacity属性从1变动至0的动画。总是这么做会很无趣。opacity只是在显示或隐藏内容时要设置动画的基本属性，并不是唯一属性。可以灵活选择动画的属性，例如通过缩小元素让其显示在视图内、通过向上滑动而划出视图，或通过改变元素的背景颜色实现。在向动画添加更多的属性时，请考虑使用多步效果，你会从下一个技巧中学到相关的知识。

3.3.5 将动画拆分为多步

要让动效设计显得专业，最简单的方法就是把动画拆分为多步效果。例如，如果通过将opacity属性和scale属性从0变动至1来显示一幅图片，那么请考虑先设置元素的opacity属性从0变动至0.5（也就是最终值的一半）的动画，然后设置scale属性从0变动至1的动画，同时设置opacity属性从0.5变动至1的动画。像这样根据属性将动画拆分为多步，就避免了网络上大多数业余动效设计中常见的线性变化的感觉，因为在那些动画中，所有属性都完全是同时进行的。在现实世界里，处于运动中的物体的各个属性并不都是同步加速的：想像一下当小鸟在飞的时候，它是怎样运动的。它向前移动（translateX）的速度与它上下移动（translateY）的速度是不同的。如果从X轴和Y轴上同时线性运动，那么看上去就更像一颗子弹而不是一只动物了。优秀的动效设计借鉴的是有生命、可呼吸的对象的运动方式，这是因为UI是为人类（而不是机器）设计的，而人喜欢感受情感和反应。

如果留心电影中所描绘的未来主义的UI动画，你会注意到那些复杂的多步动画是使它们看上去如此平滑流畅的关键。原因很简单，而且这也进一步说明了为什么要避免线性感觉：人类被多样性和反差所吸引。想一下，在拆分一个动画的时候，是怎样将该动画的每个部分放在一起对比的。就好像把层层衣服叠在一起，创造出愉悦的色彩和触感组合，也应该把动画叠加在一起，创造愉悦的动效设计组合。

想了解关于实施多步效果的技巧，请阅读第4章。

3.3.6 错开动画

如果有多个兄弟元素需要在同一时间显示出来，例如图库中的一系列图片，请考虑为它们依次添加短暂的延迟。（也就是说，在第一幅图片加载后和第二幅图片加载前出现延迟，然后以此类推。）像这样延迟兄弟元素的动画被称为*错开动画*（staggering），其目的与前面讲到的拆分动

画类似：如果所有元素都完全同步运动，会给人以呆板生硬的感觉。想要避免这种情况，可以使用错开动画在视觉上添加层次感。为什么会这样？因为同步运动一系列元素缺少类似颗粒的灵活与渐变的感觉。试想一下，鸟儿飞翔的时候不会并排成一条直线，它们的飞行看起来如此优美就是因为它们逐次改变队形，错开了时间。正是它们的编队及其动态变化使它们在人类眼中如此优雅。

3.3.7 从触发元素处产生动画

如果点击按钮会让一个模态窗口显示出来，那么请让窗口从这个按钮的位置以动画方式显示出来。换言之，让动画从触发它们的元素那里产生出来。这么做会在UI中连接因果关系，使每个元素都适得其所。

为了理解这一技巧带来的心理上的舒适感，请考虑真实世界中的运动：在拉杠杆的时候，一系列机械零件导致与之相连的物体开始运动。在这里，“相连”是关键词，真实物体只有在有力量作用在它上面时才会运动。同理，在UI中，要把每个元素视为可以施加自身力量的元素来对待，这一点非常重要。每个动作都应感觉与一个触发器相连。正是这种无缝的感觉帮助界面超越了数字世界，来到物理世界。界面越符合物理原理，就越给人有回应、有情感的感觉。

3.3.8 使用图形

利用可缩放矢量图形（SVG）可设置可供交互的较大元素的各个图形组件的动画（详见第6章）。典型的例子就是三条横线摞在一起构成的表示菜单的“汉堡”图标，或者在加载指示器中形成一个圆圈的圆点。在这些例子里，任意图形可以组合在一起构成一个常见的UI组件。在网页能够实现SVG动画以前，要想设置上述两种图形的动画，常见做法是把整个图形做在一个PNG图片里面，将该图片嵌入网页，然后通过改变CSS中的opacity属性值来设置整个图片的动画。然而，通过利用SVG，可以设置这些独特元素中的每一个形状的动画，还可以设置其多个属性的动画。

图形运动无疑会为UI中的核心位置带来别具一格的风采，其中一部分原因是由于基于网页的动画主要操作的是实心长方形。（尽管它们有时候有圆角，但形状仍保持实心和完整。）然而，通过设置元素中的各个形状的动画，可能会以出乎想像的动效设计，让用户心满意足。

除了新奇效果以外，还可以别具一格地利用SVG动画将一种形状变换为另外一种。把这种技巧与预览结果和从触发元素处产生动画结合使用，就可以利用图形变换来表示UI行为以及为用户提供的反馈。例如，如果鼠标悬停在加载指示器上面的圆点上，这些圆点就会重新排列形成一个

X的形状，这就表示用户点击这个状态指示图形就会取消加载内容。



反复实验

为每个动画确定正确的持续时间、错开动画、缓动效果和属性组合，这**并非**设计师生来就具备的技能。这一技能是每一位优秀设计师必须学习的。因此，请记住：你第一次尝试的动画属性组合，看起来可能还不错，但很可能还不是最佳效果。要想找到最佳效果，只有两种方法：一是反复实验，系统性地更改动效设计中的每一个因素，直至突然撞到一个超凡脱俗的效果；二是从别人的作品中借鉴做法。一旦找到一个喜欢的效果，还是要**进一步**实验，即使你已经一丝不差地实现了别人的这个效果。考虑是否要将持续时间减半，彻底换一种缓动类型或者更换一个属性。

设计师经常反感反复持久的实验，因为即使要把一个按钮动态显示出来也有上百万种做法，而每种做法都能有效地实现手头的目的：让按钮显示出来。结果，一旦你碰到一个属性组合构成的动画，效果还不赖，就很可能持续用它，因为看上去不错而且也能发挥作用。但请不要忘记，“不差”并不是一个值得尊敬的设计目标，“卓越”才是。要想达到卓越的水平，必须踏出舒适区，不要总是依赖于已经知道的那些起作用的设计。

3.4 小结

实用和优雅是你的目标。所有动画代码最低限度也要达到其中之一。

当运用得当时，动画代码会为UX带来实实在在的价值，而且不会对网站的性能带来负面影响。不论动效设计是多么顺滑，如果运行动画时界面显示得磕磕绊绊，那么整个用户体验也不会优雅。在第7章中，你会了解到性能的重要性。

第4章 动画工作流

大多数网站上的动画代码简直是一团乱麻。如果说经验老道的动效设计师对曾经那个古老、丑陋的Flash时代还有什么留恋的话，那就是动效设计的结构化方式了。

目前想要结构化动画代码，有两方面的做法：一是利用动画引擎（此处利用Velocity.js）的工作流特性使代码更简洁达意；二是使用代码组织最佳实践使今后的修改工作变得更容易。

不要再深层嵌套JavaScript回调函数，也别再用笨拙的CSS动画来污染样式表，跟这些做法说拜拜吧。是时候给你的网页动画设计开发升升级了。

4.1 CSS 动画工作流

为了能够更好地管理UI动画工作流，开发人员有时会放弃JavaScript而转用CSS。但是，一旦动画的复杂度达到了中等或更高水平，那么使用CSS动画通常会使工作流变得明显比用JavaScript更加糟糕。

4.1.1 CSS的问题

尽管在样式表中少量使用CSS过渡效果是很方便的，但当实现复杂的动画序列时（例如，当要求所有的元素在页面加载时顺次加载显示出来时），CSS动画代码就难以控制了。

CSS试图利用关键帧来解决这一问题，而关键帧就是让你把动画逻辑按照各自独立的时间段分离开来。

```
@keyframes myAnimation {
  0% { opacity: 0; transform: scale(0, 0); }
  25% { opacity: 1; transform: scale(1, 1); }
  50% { transform: translate(100px, 0); }
  100% { transform: translate(100px, 100px); }
}
#box { animation: myAnimation 2.75s; }
```

以上代码设置了动画时间线上几个分隔点处的特定属性应达到的值。然后，将该动画赋给ID为#box的元素，并且设置了整个关键帧序列全部完成的时间。如果没有完全明白上面的语法，不必忧心，在本书中不会用到它。但在继续往后看之前，请先考虑这样一个问题：当客户要求你把不透明度（opacity）动画时长增加1秒，但是其他属性的动画持续时间不变，会发生什么情况？要想实现这一需求，必须重新进行运算，才能使各百分比处的属性值与增加1秒后的时长匹配起来。这绝不是小事一桩，而且如果面对大量这种需求，肯定是难以应对的。

4.1.2 什么时候用CSS比较明智

有必要说明，在实现UI动画时有一种情况应当使用CSS而不是JavaScript，那就是当用户鼠标悬停在元素上时触发的简单样式变化。CSS中的transition可以极好地实现这种类型的微互动，使你仅用几行易维护的代码就能完成任务。

使用CSS时，先为目标元素定义一个transition值，这样在特定CSS属性发生改变时，动画

就会持续预先设定好的时长。

```
/* 当这个div的color属性变化时，动画持续时长为200毫秒 */
div {
  transition: color 200ms;
}
```

然后，按照每一条transition规则，设置每一个特定的CSS属性应当变动到的值。在此以hover为例，当用户鼠标悬停在div上时，它的文字颜色将会变成蓝色。

```
div:hover {
  color: blue;
}
```

这样就完成了。仅仅几行代码就可以让CSS为你处理交互状态：当用户将鼠标从div上移开时，CSS就会用200毫秒的时间，将文本颜色从蓝色再变动回之前的颜色。

好代码长什么样？

好代码是表意性的，就是说代码的目的很容易理解。这一点对同事、对自己都至关重要，因为不仅可以方便同事整合你写的“外来”代码，还能在今后忘记原本使用的方法时帮你一把。好代码同时还是简洁的，就是说用尽可能少的代码行数来完成任务；每一行都有重要目的，不能被随意改写。最后，好代码还应是易于维护的，就是说每个单独的部分都可以更新，不需要害怕影响整个程序。

相比之下，要实现上面同样的效果，jQuery的代码要写成下面这样。

```
$div
// 为div注册一个mouseover事件，调用一个animation函数
.on("mouseover", function() {
   $(this).animate({ color: "blue" }, 200);
})
// 当用户鼠标从元素上移开时，将文本颜色变动回黑色
.on("mouseout", function() {
   // 注意：我们必须记住原来的属性值（黑色）才行
   $(this).animate({ color: "black" }, 200);
});
```

上面代码虽然看起来不那么糟糕，但它没有充分发挥JavaScript提供的无限逻辑控制的优势。它费劲地做了原本CSS应该做的事，即在用户交互的同一元素上触发无需逻辑控制的动画。原本

可以在CSS中用更少、更表意、更易维护的代码实现的事情，却用JavaScript实现。更糟的是，虽然使用了JavaScript的功能，但却没有从中得到任何额外好处。

简言之，如果能很容易地用CSS的transition来设置一个元素的动画，并且该元素从未通过JavaScript来设置动画（意思是指两者之间不存在潜在冲突），那么就应该用CSS来编写该动画的代码。对于所有其他UI动画人物，例如多元素和多步骤的动画序列、交互拖拽动画等，用JavaScript实现则是上选。

让我们继续探索JavaScript提供的绝佳工作流技巧吧。

4.2　代码技巧：将样式与逻辑分离

这第一条技巧会为工作流带来巨大益处，尤其是对于团队而言。

4.2.1　一般做法

在jQuery动画中，常见的做法是通过使用UI扩展插件（jQueryUI.com）在元素上设置CSS类的动画。当该模块加载后，jQuery的`addClass()`函数和`removeClass()`函数都升级为支持动画效果。例如，在样式表中定义了下面这样一个类。

```
.fadeInAndMove {
  opacity: 1;
  top: 50px;
}
```

然后，可以在目标元素上设置这个类的CSS属性（此处是`opacity`和`top`）的动画，同时可以规定动画的持续时间。

```
// 设置.fadeInAndMove类的属性的动画，持续时间为1000毫秒
$element.addClass("fadeInAndMove", 1000);
```

另一种更常用的jQuery动画是用第1章中使用的语法，将想要设置动画的属性写到`$.animate()`调用中。

```
$element.animate({ opacity: 1, top: 50 }, 1000);
```

以上两种操作方式的结果是一样的。不同之处在于两者的逻辑分离方式不同。第一种方式将

样式规则写在了CSS样式表中，而CSS样式表正是该页面所有其他样式规则所在的地方。第二种做法将样式规则和负责触发动画的JavaScript逻辑混在了一起。

第一种方法更值得推荐，因为它的代码组织得清楚灵活，要想适当更改样式就去看样式表；要想适当更改逻辑就去看JavaScript。CSS样式表的存在是有原因的：经验丰富的开发人员不会在HTML中添加行内样式，因为那样做会混淆HTML（负责结构）和CSS（负责样式）的目的，极大地增加了网站的维护难度。

每当团队协作时，开发人员和设计人员经常头碰头地尝试同时修改同一文件。这种情况下，将逻辑分离出来的价值就愈发凸显出来。

4.2.2 优化做法

在回顾了不恰当的一般做法以后，让我们看一看优化的做法。将处理动画样式的逻辑全部写到一个专门的JavaScript文件中（如style.js），而不是写到专门的CSS样式表中，这么做大有益处，而且经常是以JavaScript为中心的动画工作流的最佳方法。听起来很奇怪，对不对？也许吧，但是这么做的效果非凡。这一技术利用简单传统的JavaScript对象帮助你组织动画代码。

举例来说，你的style.js文件可能会像下面这样。

```
// 这个对象与前面代码示例中定义的CSS类等同
var fadeIn = {
     opacity: 1,
     top: "50px"
  };
```

你的script.js是用来控制动画逻辑的主要JavaScript文件，在里面加上下面这么一行。

```
// 将已命名的属性对象传入Velocity
$element.velocity(fadeIn, 1000);
```

再重述一下技术要点：在你的style.js文件中，定义了一个JavaScript对象，该对象的内容是你想要设置动画的CSS属性。然后将这同一个对象作为第一个参数传到Velocity里。你不需要在这里做什么复杂的事情，只是把对象保存到已命名变量，然后将这些变量（而不是将原始对象本身）传到Velocity中。

注意 这一技术也同样适用于jQuery的animate()函数。

一个毫无痛苦的工作流是至关重要的。

使用JavaScript而不是CSS来分离逻辑的好处在于style.js文件具备一个独有的能力，那就是定义动画选项，而不仅仅是定义动画属性。指定某个选项有多种方式：其中一种是将两个成员属性指定给一个父级动画对象，并且你要为该父级动画对象取一个表意的名字。对象上的第一个属性用于定义动画的属性；第二个用于定义动画的选项。

这样，你的style.js文件就会像下面这样。

```
var fadeIn = {
    // p代表“属性（properties）”
    p: {
        opacity: 1,
        top: "50px"
    },
    // o代表“选项（options）”
    o: {
        duration: 1000,
        easing: "linear"
    }
};
```

在script.js文件中，这样写代码。

```
// 将我们清楚明了又可复用的动画对象传入Velocity函数
$element.velocity(fadeIn.p, fadeIn.o);
```

代码既漂亮又干净，对不对？浏览代码的人马上能明白它的意图，而且知道去哪里修改动画属性，那就是到style.js文件中去找。另外，这个动画的目的也一目了然：因为你为动画对象起了恰当的名字，因此你知道这行代码是要将一个对象淡入（fadeIn）到视图中。再也不用在脑子里分析那些动画属性来判断动画的目的是什么了。

这种做法不鼓励为一个页面上的各个动画随意设置选项，因为现在有一堆预制的动画对象可以轻易选用。结果就是代码更精简、动效设计更一致。正如在前一章中所学到的，一致性是优秀用户体验的重要特征之一。

这种做法的好处不止于此，最棒的是它能够完美组织动画的各种变体。例如，通常情况下会让按钮元素在1000毫秒内淡入，但对于模态窗口却希望用3000毫秒来淡入。这时候，只需要把选项对象一分为二，分成两个恰当命名的变体即可。

```
var fadeIn = {
     p: {
        opacity: 1,
        top: "50px"
     },
     // 选项对象变体一： 持续时间较快
     oFast: {
       duration: 1000,
       easing: "linear"
     },
     // 变体二：持续时间较慢
     oSlow: {
       duration: 3000,
       easing: "linear"
     }
  };
// 设置动画使用较快的持续时间
$button.velocity(fadeIn.p, fadeIn.oFast);
/* 设置动画使用较慢的持续时间 */
$modal.velocity(fadeIn.p, fadeIn.oSlow);
```

另一种做法是将fast和slow作为子对象嵌套在一个单独的o选项对象里面。两种做法喜欢用哪

一种取决于你的个人喜好。

```
var fadeIn = {
    p: {
      opacity: 1,
      top: "50px"
    },
    o: {
      fast: {
        duration: 1000,
        easing: "linear"
      },
      slow: {
        duration: 3000,
        easing: "linear"
      }
    }
  };
// 设置动画使用较快的持续时间
$button.velocity(fadeIn.p, fadeIn.o.fast);
/* 设置动画使用较慢的持续时间 */
$modal.velocity(fadeIn.p, fadeIn.o.slow);
```

如果感觉这么写太麻烦，而且你的动画逻辑本来就没有几行JavaScript代码，那么直接在调用时设置属性和选项也没问题。不要因为把这个步骤完全省去了就感觉自己是个差劲的开发人员。你永远都应该使用适合自己项目的方法，不论这种方法多么适合其他项目。这里学到一个简单的道理：如果发现自己需要动画工作流的最佳实践，那么它就确实存在。

4.3 代码技巧：组织排序动画

Velocity有一个名为UI pack的小插件。（请至VelocityJS.org/#uiPack获取。）它大幅优化了UI动画的工作流，从而提升了Velocity的功能。本章中的许多技巧，包括下面将要讨论的，都用到了这个插件。

要安装UI pack，只需要在Velocity的<script>标签后面，页面上body结尾处的</body>标签前面为它加上<script>标签。

```
<script src="velocity.js"></script>
<script src="velocity.ui.js"></script>
```

本节中要讨论的这个UI pack功能叫作*顺序运行*（sequence running）。该功能将会从此改变你的动画工作流。它是治愈胡乱嵌套的动画代码的灵丹妙药。

4.3.1 一般做法

如果不使用UI pack，要连续设置不同元素的动画的一般做法如下。

```
// 设置element1的动画，紧接着设置element2的动画，紧接着设置element3的动画
$element1.velocity({ translateX: 100, opacity: 1 }, 1000, function() {
  $element2.velocity({ translateX: 200, opacity: 1 }, 1000, function() {
    $element3.velocity({ translateX: 300, opacity: 1 }, 1000);
  });
});
```

不要让这个简单的例子蒙蔽你的双眼：在真实的产品代码中，动画序列里包含比上面例子中多得多的属性、选项和嵌套关系。这样的代码最常出现在加载序列（当某个页面或某个子部分首次加载时）中，因为它需要设置多个元素到位的动画。

注意上面显示的代码不同于将多个动画链到同一元素上的操作，后者毫不麻烦而且也不需要嵌套。

```
// 将多个动画链到同一元素上
$element1
  .velocity({ translateX: 100 })
  .velocity({ translateY: 100 })
  .velocity({ translateZ: 100 });
```

那么第一段代码示例 （就是设置不同元素的动画的代码示例 ）究竟哪里出了问题？此处可以列举几条主要问题。

- 因为每一级都有嵌套，因此代码在横向上会迅速变长，这使得在IDE中编辑代码越来越困难。
- 在整个序列中，都无法轻易重新调整调用的顺序。（如果要调整的话，需要非常小心地复制粘贴。）
- 无法轻易指明哪些调用应该是并行运行的。比如在整个序列运行到一半的时候，想让两幅图片从不同的起始点滑入视图。把这段代码写进去以后，怎么样嵌套并行的小序列之后发生的动画，同时又不让原本已经难以维护的代码变得难上加难，就不是那么轻易能办得到了。

4.3.2 优化做法

在学习解决这个丑陋问题的优雅方式之前，很重要的一件事是理解Velocity的两个简单功能。首先，要知道Velocity接受多参数语法：最常见的写法就是在jQuery元素调用Velocity时，参数包括一个属性对象，后面跟着一个选项对象（之前所有的代码示例都是这么写的）。

```
// 迄今为止使用的参数语法
$element.velocity({ opacity: 1, top: "50px" }, { duration: 1000,  easing: "linear" });
```

另一种语法则要与Velocity的*效用函数*（utility function）配合使用，该效用函数可以使用基本的Velocity对象设置元素的动画，而不是断开与一个jQuery元素对象的链接。以下就是设置离开基本Velocity对象的动画的代码写法。

```
// Velocity将自己注册到jQuery的$对象上，可以在此处使用
$.Velocity({ e: $element, p: { opacity: 1, scale: 1 },  o: { duration: 1000, easing: "linear" } });
```

正如上述代码显示的，另一种写法就是将*一个单独的对象*传入Velocity函数，该单独对象中的成员对象一一映射到每一个标准Velocity参数上（*元素*elements、*属性*properties和*选项*options）。为了简便起见，每个成员对象的名称只用了对应对象的首字母来表示（e代表elements、p代表properties、o代表options）。

另外，注意此处是将目标元素作为Velocity的一个参数传进去的，因为你不再是直接在一个元素上触发Velocity。最终效果与之前的那种写法是一模一样的。

如你所见，新写法并没有让代码变得笨重许多，但至少同样利于表意。知道了这种新写法，就可以学习使用UI pack的顺序运行功能了：只需要创建一个由Velocity调用构成的数组，每一个调用都使用单个对象的写法，请参考下面的代码示例。然后，将整个数组传入一个特殊的Velocity函数中，该函数可以连续触发调用序列。当一个Velocity调用完成以后，下一个就会执行，即使每个调用针对的是不同的目标元素也是如此。

```
// 创建 Velocity 调用的数组
var loadingSequence = [
    { e: $element1, p: { translateX: 100, opacity: 1 },  o: { duration: 1000 } },
    { e: $element2, p: { translateX: 200, opacity: 1 },  o: { duration: 1000 } },
    { e: $element3, p: { translateX: 300, opacity: 1 },  o: { duration: 1000 } }
];
// 将数组传入 $.Velocity.RunSequence 函数中以启动序列
$.Velocity.RunSequence(loadingSequence);
```

这么做的好处很明显。

- 可以很容易地对整个序列重新排序，不需要担心破坏了嵌套结构。
- 可以迅速看出各个调用的属性和选项的不同之处。
- 你的代码对别人来说也非常易读，其意图也一目了然。

如果将这一技巧与前面讲到的技巧（把CSS类写成JavaScript对象）结合使用，代码看上去就显得格外优雅了。

```
$.Velocity.RunSequence([
  { e: $element1, p: { translateX: 100, opacity: 1 }, o: slideIn.o },
  { e: $element2, p: { translateX: 200, opacity: 1 }, o: slideIn.o },
  { e: $element3, p: { translateX: 300, opacity: 1 }, o: slideIn.o }
]);
```

表意性和易维护性并非顺序运行仅有的好处：还可以通过使用特殊的sequenceQueue选项获得并行运行各个调用的能力。将其设置为false时，单独的调用与它之前的调用就会并行执行。这可以让多个元素同时以动画方式进入视图，并且只用一个Velocity序列就能够对时间安排进行错综复杂的控制。如果没有这种方法，通常情况下要实现这种效果需要通过乱糟糟的回调嵌套来进行精心安排才行。请参考示例中的行内注释了解详情。

```
$.Velocity.RunSequence([
  { elements: $element1, properties: { translateX: 100 }, options: { duration: 1000 } },
  // 下面一个调用会与第一个调用同时开始，因为它使用了sequenceQueue: false选项
  { elements: $element2, properties: { translateX: 200 }, options: { duration: 1000, sequenceQueue:
false },
  // 按照正常情况，下面的调用将在第二条调用完成之后执行
  { elements: $element3, properties: { translateX: 300 }, options: { duration: 1000 }
];
```

4.4 代码技巧：打包你的效果

动效设计中最常用的效果之一就是淡入淡出内容。这种动画类型经常是一系列单独的动画调用链在一起，呈现出细腻、多层次的效果。

4.4.1 一般做法

除了简单地设置一个元素的opacity值变动至1的动画以外，你可能想要同时设置元素的

scale属性的动画，让这个元素看上去不仅在淡入，而且在不断长大直至最终大小。一旦元素完全显示出来，作为收尾润色，你还可能想要设置元素边框（border）的宽度变动至1rem的动画。如果这个动画在一个页面上要发生多次，而且是在不同元素上发生的，那么为了避免代码重复，把它写成一个独立的函数就很有必要了。否则的话，就必须在script.js文件中反复重复下面这段不能表意的代码。

```
$element
  .velocity({ opacity: 1, scale: 1 }, { duration: 500, easing: "ease-in-out" })
  .velocity({ borderWidth: "1rem" }, { delay: 200, easing: "spring", duration: 400 });
```

不同于前一节中讨论的顺序运行技巧，上面这段代码包含了发生在同一元素上的多个动画。在同一个元素上链在一起的多个动画构成一个*效果*。如果想要通过本章所讲的第一个技巧（把CSS类写成JavaScript对象）来优化这个效果的话，就必须费劲地把整个动画每一阶段的每个参数对象命名。但是，一方面由于这一特定序列的独特性，这些对象可能不会在动画代码的其他地方用到；另一方面，还必须得为每个动画调用对应的对象追加整数，好让它们彼此区分开来。这样就会造成混乱，也会抵消将CSS类写成JavaScript对象这一技巧所带来的代码组织和简洁方面的好处。

像这种效果的另一个问题在于代码不能很好地表意，也就是说，其意图不是一眼就能看得出来。为什么要用两个动画调用而不是一个？为什么为每个调用选择这些属性和选项？这些问题的答案与触发动画的代码无关，因此应当被隐藏起来。

4.4.2　优化做法

Velocity的UI pack能让你注册效果，然后就可以在整站中重复使用了。一旦注册效果之后，就可以通过将注册名称作为第一个参数传入Velocity进行调用。

```
// 假设我们将效果注册在"growIn"名下
$element.velocity("growIn");
```

这样一来，表达的意图就更明显了，不是吗？你马上就能了解代码的目的：一个元素会逐渐变大显示出来。同时，代码仍保持简洁、易于维护。

另外，已注册的效果与标准Velocity调用的行为完全一致；可以像往常一样传入一个选项对象，而且也可以把其他Velocity调用链在上面。

```
$element
  // 将元素滚动到视图中
  .velocity("scroll")
  // 然后用以下设置在元素上触发"growIn"效果
  .velocity("growIn", { duration: 1000, delay: 200 })
```

如果UI pack已经加载到你的页面上，像这样的效果可以用以下写法进行注册。

```
$.Velocity.RegisterEffect(name, {
  // 未将duration值传入到调用中时，将使用的默认duration值
  defaultDuration: duration,
  // 以下Velocity调用一个接一个发生，每个所用的时间等于效果的总时间乘以预定的比例
  calls: [
     [ propertiesObject, durationPercentage, optionsObject ] ,
     [ propertiesObject, durationPercentage, optionsObject ]
   ],
  reset: resetPropertiesObject
});
```


让我们一步步解读这一模板。

(1) 第一个元素是效果的名称。如果这个效果负责的是将元素在视图中显示出来（比如在淡入效果中，将元素的opacity值从0变动至1），那么重要的一点就是在名称后面添加后缀In。

(2) 第二个参数是一个对象，它定义了效果的行为。该对象中的第一个属性是default Duration。在触发效果的Velocity调用没有传入duration时，该属性指定的就是整个效果所需的持续时间。

(3) 对象中的第二个属性是calls数组，它包含构成这个效果的Velocity调用（按照发生的先后顺序排列）。这个数组中的每一项又是一个数组，其中包含调用的属性对象；然后是该调用占用效果总时长的比例（一个十进制值，默认为1.00），这一项为可选项；最后是该调用的选项对象，该项也是可选项。注意，在calls数组中指定的Velocity调用仅接受easing和delay两个选项。

(4) 最后，可以选择传入一个reset对象。该reset对象的写法与标准的Velocity属性映射对象的写法一样，但它是用来在效果全部完成之后立即更改属性值的。当你设置了一个元素的opacity和scale属性值变动至0的动画（离开视图），在元素隐藏后又想将元素的scale属性重新设为1时，重置对象就很有用了。这么做可以让后续效果不需要担心多余的事情，即除了opacity以外还要把其他属性重置才能让效果正确生效。换言之，可以利用reset属性映射让这个效果自成一体，不在目标元素上留下扫尾工作。

除了reset对象以外，UI pack的效果注册还为工作流带来了另一强大功能，那就是自动的display属性切换。当一个元素开始以动画方式进入视图时，你想要确保display的值设为none以外的其他值，这样元素在整个动画过程中都是可见的。（请记住，display: none将会把元素从文档流中移除。）反过来，当一个元素淡出时，你通常想要确保opacity值达到0时，将display的值改为"none"。这么做，就可以在用完这个元素之后移除它所有存在的痕迹。

使用jQuery时，display切换是通过将show()和hide()辅助函数链到动画上来完成的（经常会被乱糟糟地埋在嵌套回调中）。然而使用Velocity的UI pack，会自动实现这一逻辑，只要你在给效果命名时恰当地添加了In或Out作为后缀。

让我们注册两个UI pack效果：一个方向是In，另一个方向是Out。之所以称效果为shadowIn是因为它可以让元素淡出、变大显示，然后将其boxShadow属性向外扩大。

```
$.Velocity
  .RegisterEffect("shadowIn", {
     defaultDuration: 1000,
     calls: [
         [ { opacity: 1, scale: 1 }, 0.4 ] ,
         [ { boxShadowBlur: 50 }, 0.6 ]
     ]
  })
  .RegisterEffect("shadowOut", {
     defaultDuration: 800,
     calls: [
        // 我们反转顺序，以镜像"In"方向
        [ { boxShadowBlur: 50 }, 0.2 ],
        [ { opacity: 0, scale: 0 }, 0.8 ]
     ]
  });
```

如果将效果名称以Out结尾，Velocity会自动在动画结束后，将元素的display属性设为none。反之，如果效果名称以In结尾，Velocity会自动将display属性设置为与元素标签类型匹配的默认值。（例如，为链接设置的值为"inline"；为div和p设置的值为"block"。）如果效果名称中不包含这两个特殊后缀中的任意一个，那么UI pack将不会执行自动display设置。

注册效果不仅优化了代码，而且使其变得非常易于用于其他项目中或供其他开发人员使用。当你设计了一个很喜欢的效果，现在就可以轻松地将效果的注册代码分享给他人使用。真是非常地便捷！

4.5 设计技巧

本章迄今为止所讨论的技巧都是在动效设计的编程阶段提升工作流。本节中讲述的这些技巧则注重设计环节，也就是仍在试验以求找到适合UI的完美动画的阶段。这一阶段需要很多创造力和重复劳动，因此相应地也需要工作流的提升。

4.5.1 定时乘数

第一个设计技巧是使用全局定时乘数，即在动画中所有的delay和duration值处都添加一个恒定的乘数。

首先定义你的全局定时乘数（强制指定M作为乘数）。

```
var M = 1;
```

然后，在每一个动画调用中，将这个乘数与duration和delay选项值相乘。

```
$element1.animate({ opacity: 1 }, { duration: 1000 * M });
$element2.velocity({ opacity: 1 }, { delay: 250 * M });
```

注意 如果你使用SASS或LESS，由于这两者都支持在样式表中使用变量，因此这一技巧也同样适用于CSS动画！

嵌入乘数常量可以帮助你在一个地方（可能在style.js文件的顶部）迅速修改M常量的值，从而能马上加快或减慢整个页面上的所有动画。这种时间控制的好处包括下面两个。

- 将动画的速度减慢，有助于在复杂动画序列中完善单个动画的调用时间。当你为了优化包含多个元素的动画序列而要反复刷新页面时，让这个序列慢速播放可以更容易地分析单个元素如何与其他元素互动。
- 执行重复性UI测试时，请加快动画播放速度。要测试网站的其他方面而不是动画时，评价UI动画的最终状态（元素最终是什么样）比测试动画的过程更加重要。在这种情况下，这么做可以节省时间，不需要头疼如何去加速动画，从而减少每次页面刷新后等待动画播放完成的时间。

该功能可以很方便地通过Velocity中一个名为mock的东西实现。mock作为幕后的全局乘数发

挥着作用，不再需要手动在各处添加M常量了。就像下面示例中显示的，mock会同时乘以duration和delay值。要想启用mock，可以临时将$.Velocity.mock设为你想要的乘数值。

```
// 所有动画的时间都乘以5
$.Velocity.mock = 5;
// 现在所有动画的时间都是经过调整的
// 下面的duration实际上是5000毫秒
$element.velocity({ opacity: 1 }, { duration: 1000 });
```

Velocity的mock功能同样接受布尔值：将mock设为true就会将所有的持续时间和延迟时间设为0毫秒，这会强制所有动画在浏览器计时中断一次的时间内完成，这样的中断每几毫秒就会发生一次。这是一种强大的简便方法，能让你在动画妨碍到UI的开发和测试时，迅速关闭所有动画。

4.5.2　使用Velocity动效设计器

制作Velocity动效设计器（VMD）的唯一目的就是帮助开发人员流畅地进行动效设计创作。VMD是一个书签工具，你可以把它加载到页面上从而实时地进行动画设计。它还允许你通过双击元素打开一个模态窗口，在该窗口内可以为该元素设置动画的属性和选项。然后，在键盘上点击回车就可以马上查看动画效果，而无需刷新页面。

注意　可前往http://velocityjs.org/#vmd获取Velocity动效设计器。

一旦所有元素的动画都已经设计为想要的样子，就可以把成果导出成一对一的Velocity代码，然后可以马上把这段代码放到IDE中用于制作。（生成的代码与jQuery也是完全兼容的。）

最后，VMD避免了持续不断的IDE和浏览器标签页之间的切换以及重复触发UI状态，因此节省了不计其数的开发时间。另外，它使设计师至开发人员的工作流更为流畅，因为它允许两个团队实时并肩工作：通过使用VMD，设计师可以在不熟悉网站的JavaScript和CSS的情况下也能进行动效设计。他们只需要把导出的Velocity代码传给开发人员，然后让开发人员酌情整合到代码库里面即可。

让动效设计
变得
趣味横生。

VMD是一个高度视觉化的工具，请访问VelocityJS.org/#vmd观看介绍视频。

4.6 小结

当使用动画工作流技巧时，你会注意到动效设计那令人生畏的黑匣子已经开始慢慢打开了。像Stripe.com和Webflow.com这样的前沿网站上看到的精巧漂亮的加载动画，你已经开始能看懂了。你能够编写动画序列的代码，这将给你信心。同时，这些新学到的技能将会减少你在开发中的繁琐工作，使实现动效设计目标变得不仅更简单，而且也有趣得多。

第5章 文本动画

邮

电

鉴于网页上很少使用文本动画，因此它成为了一种打动用户的简单方式。恰恰由于这个原因使这个主题学起来趣味横生：基本的技术很容易掌握，但效果在用户看来却异常丰富、复杂。

本章介绍了几种工具，可以帮你免于文本动画枯燥乏味的一面，并且为你提供了高效的工作流。继续往下读，学习这个黑魔法的玄妙之处。

5.1 文本动画的一般做法

我们通常编写网站使用的HTML元素（例如div、table、anchor标签和其他类似元素）都是网页上能够添加样式的最低一级的组成部分。因此，我们自然而然认为这些也是能够添加动画的最低一级组成部分。

文本本身并不构成一个元素；浏览器会指定一块文本作为一个文本节点（text node），而文本节点是没法添加样式的，它是更低一级的网页组成，必须被包含在一个元素里才行。另外，浏览器不会将文本节点再细分为语法成分；你没有办法操作单个字母、单词或句子，这使得问题更为棘手。

因此，要想以字母、单词或句子为单位设置文本的动画，必须要把每个文本节点以此为单位拆分成单独的文本节点，然后把它们每一个再用一个新元素包起来。这样就可以设置其动画了。但是，手动将文本包裹在例如span元素内，这一工作不仅枯燥乏味，而且还会让HTML臃肿不堪。

这么说来，网络上鲜见文本动画就不足为奇了，因为总是会面临很多麻烦。如此一来，网络相对于Adobe After Effects这样的专业动效设计软件就处于审美方面的劣势，后者可以设计出精细的文本动画，这样的动画在电视广告或电影片头经常会看到。这些效果看上去真是无比精美。但可惜的是，它们不仅很难被整合到网络上，而且即使整合了也会被广泛认为是糟糕的实践方式。毕竟，网络是注重功能大于形式的媒介，而文本动画主要还是一种形式。

但是，有一种文本动画使用案例，如果谨慎使用的话，是可以很好地移植到网络上的：如果仔细注意过电影中对未来硬件界面的描绘，就会注意到常见的文本主线是根据语法规则以动画方式进入或离开视图的。根据流行文化所展示的，未来的电脑科技中包含单词和句子的动画，例如闪烁、毛刺（glitch）、弹出和模糊效果。这些效果很酷，而且把它们用于动态显示隐藏文本也没多大坏处，因为文本总归要用到这样或那样的调整可见性的动画的。这种动态改变文本可见性的思想正是你在本章将要学到的。

文本动画
看起来
就是酷。

5.2　为使用 Blast.js 实现动画准备文本元素

此处选择的实现文本动画的工具是Blast.js，它可以很方便地将文本块拆成字符、单词和句子。然后你就可以用Velocity和UI pack插件来给这些拆开的部分加动画了。

注意　前往Julian.com/research/blast获取Blast.js。

Blast.js有三种分隔符（delimiter）类型，分别定义了三种单独提取的语法成分：字符、单词和句子。假设你有一个下面这样的div。

```
<div>
  Hello World
</div>
```

如果你通过下面的语法调用Blast函数

```
$("div").blast({ delimiter: "word" });
```

这个div就会变成下面这个样子。

```
<div class="blast-root">
  <span class="blast">Hello</span>
  <span class="blast">World</span>
</div>
```

正如你所见，Blast将目标div的文本分割成一个个由span元素包裹起来的文本部分。如果改而使用character分隔符，那么结果就会是下面这个样子。

```
<div class="blast-root">
  <span class="blast">H</span>
  <span class="blast">e</span>
  <span class="blast">l</span>
  <span class="blast">l</span>
  <span class="blast">o</span>
  <span class="blast"> </span>
  <span class="blast">W</span>
  <span class="blast">o</span>
  <span class="blast">r</span>
  <span class="blast">l</span>
  <span class="blast">d</span>
</div>
```

现在你就可以单独设置这些span元素的动画了。但是，在深入了解文本动画之前，应当多学习一下Blast的工作原理，这样才能够充分利用它的强大功能。

5.2.1　Blast.js的工作原理

本节的目的就是让你放心地使用Blast将心爱的页面上的文本进行拆分。让我们继续！

你所熟悉的div、table和其他HTML元素都被称为元素节点（element node）。一个元素节点通常包含两种类型的子节点：其他元素节点和文本节点（原始文本）。

以下面这个元素为例。

```
<div>
  Hello <span>World</span>
</div>
```

这个div元素包含两个子节点：一个文本节点（Hello）和一个span元素节点。span元素节点又有一个自己的子节点：另一个文本节点（World）。

调用Blast时，Blast会遍历目标元素的整个后代元素链，以找出文本节点。对于每一个文本节点，Blast依据指定的分隔符（`character`、`word`或`sentence`）执行正则搜索，将这个文本节点拆分为新的元素，每个元素中有它自己的那部分文本节点内容。由于Blast实际上不拆分*元素*节点，而只拆分文本节点，因此，可以安全地将其用于整个页面，不需要担心会破坏原本元素的事件句柄（event handler）或发生其他不可预测的行为。将Blast用于经常夹杂着HTML代码的用户生成的内容时，这种通用性就至关重要了。（例如，你想要把发布在网站评论区的一条信息按单词拆分，以便可以强调重点段落。这时使用Blast来拆分就很安全，你不需要担心破坏用户的嵌入式链接。）

除了抗扰性以外，Blast还提供了高水平的准确性。它不是傻傻地在空格处拆单词，或在句号处拆句子。它利用的是拉丁字母语言的UTF-8字符集，也就是说可以把它用在法语、德语、西班牙语、英语、意大利语和葡萄牙语的内容上，得出的结果依然正确。

5

假设你在下面的段落上使用了Blast的`sentence`分隔符。（下面所使用的**粗体**和*斜体*代表Blast检测出来的连续匹配文本。）Blast正确地识别出了段落中的六个句子。

¿Will the sentence delimiter recognize this full sentence containing Spanish punctuation? *¡Yes!* **« Mais, oui ! » “Nested “quotes” don’t break the sentence delimiter! ”** *Further, periods inside words (e.g. Blast.js), in formal titles (e.g. Mrs. Bluth, Dr. Fünke), and in “e.g.” and “i.e.” do not falsely match as sentence-final punctuation.* **Darn.** *That’s pretty impressive.*

请注意标点符号跟在了正确的句子后面，不规则的句点也没有错误地把整句拆散。

了解了Blast的这些基本功能后，是时候看看Blast的使用方法了。

5.2.2 安装

Blast在页面上的安装方式就像其他任何JavaScript插件一样：将合适的script链接嵌入到页面

</body>标签之前。

```
<html>
  <head>My Page</head>
  <body>
    My content.
    <script src="jquery.js"></script>
    <script src="velocity.js"></script>
    <script src="blast.js"></script>
  </body>
</html>
```

注意　Blast需要使用jQuery（也可用Zepto替换jQuery），因此必须将它添加在jQuery的标签之下。不论Blast是在Velocity之前还是之后加载，都没有关系。

一旦Blast加载之后，就可以通过在jQuery元素对象上调用.blast()来使用它了。Blast只接受一个选项对象作为其唯一参数。

```
$element.blast({ option1: value1, option2: value2 });
```

下面让我们逐一看一下可用的选项。

5.2.3　选项：delimiter（分隔符）

Blast最重要的选项就是delimiter，它接受"character"（字符）、"word"（单词）或"sentence"（句子）。要使用"sentence"分隔符对$element中的文本进行拆分，你的代码会是下面这个样子。

```
$element.blast({ delimiter: "sentence" });
```

请注意，Blast把生成的文本包装器元素返回到jQuery选择器链上，于是你就可以像下面这样来操作了。

```
$element.blast({ delimiter: "sentence" })
  .css("opacity", 0.5);
```

这里的.css()调用是针对单独的文本元素，而不是它们的父级元素，即调用Blast的$element元素。

5.2.4 选项：customClass（自定义类）

为了操作更为简便，Blast提供了两个选项：customClass和generateValueClass。customClass的行为应该与你料想的一模一样：将一个自定义类（作为字符串）指派给文本节点包装器元素。

假设你有下面这样一个div并调用了Blast。

```
<div>
  Hi Mom
</div>
$("div").blast({ delimiter: "word" , customClass: "myClass" });
```

这个div就会变成下面这个样子（注意Blast会自动给每个文本部分默认添加"blast"类）。

```
<div>
  <span class="blast myClass">Hi</span>
  <span class="blast myClass">Mom</span>
</div>
```

提供自定义类的价值在于区分不同Blast调用所产生的元素。例如，你在页面的两个位置上使用了Blast：一个在页眉，一个在页脚，那么为这两处调用指定不同的类也许会帮助后续的JavaScript代码和CSS样式在文本元素上恰当地发挥作用。

5.2.5 选项：generateValueClass（生成值类）

generateValueClass接受一个布尔值（true或false），用来表示是否要添加一个特有的类（以.blast-[delimiter]-[textValue]的格式）到生成的文本元素上。

注意 此选项仅适用于character和word分隔符。

[delimiter]占位符代表的是调用中使用的分隔符类型；[textValue]占位符代表的是单独元素中包含的文本。请看下面的例子。

```
<div>
  Hi Mom
</div>
$("div").blast({ delimiter: "word" , generateValueClass: true });
```

这个元素会变成下面这个样子。

```
<div class="blast-root">
  <span class="blast blast-word-Hi">Hi</span>
  <span class="blast blast-word-Mom">Mom</span>
</div>
```

当Blast调用时delimiter为`letter`，那么该元素则会变成下面这个样子。

```
<div class="blast-root">
  <span class="blast blast-letter-H">H</span>
  <span class="blast blast-letter-i">i</span>
  ... and so on...
</div>
```

当你需要根据文本内容针对匹配的文本元素进行CSS或JavaScript操作时，`generateValueClass`这个选项就很有用处了。例如，如果将这个功能用在了一段图书摘录上，就可以通过给类为`.blast.word-and`的元素设置黄色背景，创建单词and的所有实例的视觉效果。

```
// 使用jQuery实现
$(".blast-word-and").css("background", "yellow");
// 使用纯JavaScript实现
document.querySelectorAll(".blast-word-and").forEach(function(item) { item.style.background =
"yellow"; });
// 使用CSS实现
.blast-word-and {
  background: yellow;
}
```

多亏这个功能，你可以轻松地通过CSS或JavaScript匹配目标文本，而无需特意使用乱糟糟的自定义代码来检查每个元素里面的文本内容。

5.2.6　选项：tag（标签）

此选项用于指定包装文本部分的元素的类型。默认值为`span`，但你也可以传入任何其他元素类型（例如：`a`、`div`、`p`）。请看下面的例子：

```
<div>
  Hi Mom
</div>
```

```
// 将"div"元素用作包装器标签
$("div").blast({ delimiter: "word" , tag: "div" });
```

该元素最终会变成下面这样：

```
<div class="blast-root">
  <div class="blast">Hi</div>
  <div class="blast">Mom</div>
</div>
```

这一功能的实用之处在于它可以确保生成的文本元素与周围html的结构相仿。也许附近的兄弟元素都是div，在这种情况下上面例子中的应用可能就比较适合。

另外，你还可能想要利用不同标签类型的特殊属性。例如，strong标签会自动让文本变成粗体，而div标签会强制每个匹配文本都从新一行开始，因为div的默认display属性值为"block"。

5.2.7 命令：reverse（反转）

我们可以在一个元素上反向执行Blast，方法是将false作为唯一参数传到Blash调用中。因此，如果你已经拆分的元素像下面这样：

```
<div class="blast-root">
  <div class="blast">Hi</div>
  <div class="blast">Mom</div>
</div>
```

然后你进行了如下的Blast调用：

```
$("div").blast(false);
```

那么元素就会变成原来的样子：

```
<div>
  Hi Mom
</div>
```

你也许好奇这究竟是如何实现的：当Blast进行反转操作时，它只是简单地摧毁生成的包装器元素，然后把纯文本插入到原有的包装器元素里面。请注意，这样做将会破坏指派给新生成元素的事件句柄，但是不会破坏Blast第一次调用之前就已经存在的HTML上的事件句柄。

这种方式的Blast反转操作对于文本动画而言至关重要，因为给网页上的元素设置动画之后，还要让这些元素恢复原样。例如，如果使用Blast将一句话拆分成单词，为的是让这些单词一个个以动画方式进入视图，那么就要在动画完成之后，接着反转Blast操作。最终，以后与文本交互的JavaScript代码就不会冒出一些出乎意料的子元素需要解析了。简言之，最佳做法是：尽量避免HTML代码变得过分臃肿，这样后续与元素的程序交互就不会变得更加复杂。

注意 要学习Blast的更多知识，包括它特有的搜索功能以及与屏幕阅读软件的适配性，请查看Julian.com/research/blast上的文档。

既然现在已经把文本元素拆开来了，是时候给它们加动画了。

5.3 让文本过渡进入视图或离开视图

文本动画最常见的用法就是让文本以动画方式进入或离开视图。其中一个基本应用就让句子中的单词一个接一个地进入视图。

5.3.1 替换已有文本

让我们先创建一个div容器，其中包含占位文本。占位文本将会被以动画方式进入视图的新文本替代：

```
<div>
  A message will load here shortly...
</div>
```

由于这个div一开始就是可见的，因此用Blast将div的文本拆成的子文本元素也是可见的。既然你的目标是让新生成的文本元素从初始状态不可见变成可见，那么就应该在调用Blast之后马上将生成的文本元素设置为不可见：

```
$("div")
  .html("This is our new message.")
  .blast({ delimiter: "word" })
    .css("opacity", 0);
    .velocity({ opacity: 1 });
```

以上代码会将div中原有的文本替换为新信息。然后用Blast以word为分隔符将div拆分开来。既然Blast往jQuery选择器链上返回的是生成的文本包装器元素，因此你可以轻易地在后面续写代码，将每个文本元素的opacity值设为0。这样就使元素做好准备工作，进而继续调用Velocity函数，其中包含一个简单的opacity动画。

你也许注意到了，上述代码让所有文本部分同时以动画方式进入视图。这种做法当然有违使用Blast的初衷：如果想让整个div的内容同时以动画方式进入视图，只需要简单地设置div本身的动画就可以了。但我们这里的目标其实是实现一个连续的动画序列，让文本元素一个接一个地以动画方式进入视图。

5.3.2 错开动画

这时候就是Velocity的UI pack登场的时候了。（如果你需要UI pack的入门说明的话，请参考第4章。）要给一个元素集合中每个元素的动画开始之间相继添加延迟，要使用Velocity UI pack中的stagger选项，它规定的是以毫秒为单位的一段时间。将其用于前面的代码示例，就得到如下代码：

```
$("div")
  .html("This is our new message.")
  .blast({ delimiter: "word" })
    .css("opacity", 0);
    .velocity("transition.fadeIn", { stagger: 50 });
```


上面的代码会在每个元素动画开始之前相继添加一个50毫秒的延迟。重要的是，请注意之前Velocity调用中的{ opacity: 1 }参数被替换为"transition.fadeIn"，这是Velocity的UI pack中预制的淡入效果。（如果你需要复习一下的话，请参考第4章。）由于stagger选项可以与UI pack效果一起使用，此示例显示了将动画opacity仅镜像至值1的效果。

正如第3章中讨论过的，要注意确保错开时间的长度很短，这样用户在文本淡入时就不会浪费不必要的时间等待。请记住，文本的单词数越多，整个动画序列完成的时间就越长。错开文本元素动画是最容易导致拖慢界面这种糟糕后果的方式之一。

5.3.3 过渡文本离开视图

上一节中的代码示例仅展示了如何让文本以动画方式进入视图，而未展示如何离开视图；div中原本已存在的文本是立即被新信息替换掉的。这种做法不一定会导致糟糕的动效设计，但从动

效设计理论的角度讲，最好是将元素淡出的动画与淡入的动画统一起来。第3章讲到了镜像动画的概念，说的就是动画怎么来就应让它怎么去。这一建议在此处依然适用。

如果想让文本离开的动画与进入的动画对称，可以重新编写代码如下：

```
// 选择之前已拆分的文本
$("div .blast").velocity(
  // 使用适当的UI pack效果让现有文本以动画方式离开视图
  "transition.fadeOut",
  {
     // 像错开进入动画一样错开离开动画
     stagger: 50,
     backwards: true,
     // 当文本离开视图的动画完成以后，开始文本进入视图的动画
     complete: function() {
        // 继续进行文本进入视图的动画
        $("div")
           .html(message)
           .blast({ delimiter: "word" })
             .css("opacity", 0)
             .velocity({ opacity: 1 }, { stagger: 50 });
     }
   }
);
```

以上代码首先在之前div拆分后生成的文本部分上调用了Velocity UI pack中的"transition.fadeOut"效果。就像进入方向的动画一样，在离开方向上，stagger选项为每个文本部分的动画相继错开一定时间。此调用中新使用了Velocity UI pack的backwards选项，该选项可以与stagger搭配使用来反转目标元素集的顺序，这样，最后一个元素（句子中的最后一个单词）比倒数第二个元素先以动画方式离开视图；而倒数第二个元素比倒数第三个元素先以动画方式离开视图，以此类推。当离开视图的动画序列完成之后，就会从complete回调中调用进入视图的动画了。

在文本动画中使用backward选项带来两点好处。首先，便于与进入方向的动画对称（创建顺序完全颠倒的动画）。进入动画中，第一个单词是先于第二个单词进入视图的，以此类推。第二，当倒序的动画后面马上接着一个正序的动画，会产生一种优雅的链式效果，倒序方向上的最后一个单词和正序方向上的第一个单词是紧接着出现的。这样就把两个动画序列绑在了一起，让人看起来好像是一个动画整体，而非两个强行粘在一起的单独动画。

5.4 过渡单个文本部分

电影字幕序列以其别出心裁的排版动效设计而闻名。它的很多效果都依赖于一种技术，那就是挑出单个文本元素以设置其动画。这也正是本节要讲到的内容。

注意 要想寻找排版动画的灵感，可以到YouTube上搜索电影字幕来看，并且记下详细笔记！只要始终牢记动效设计理论的几条原则，你应该对自己在界面上探索文本动画设计充满信心。

要想对Blast生成的元素进行精细控制，只需要使用CSS中的nth-child选择器或jQuery中的eq()函数。两者的行为基本一致，都可以让你在元素集合中，基于元素的索引值来选择某一个元素。如果将整数3传到CSS的选择器中（或将2传入jQuery函数中，正如你将要看到的），它们会在整个元素集合（即多个单词组成的句子）中锁定第三个元素（即第三个单词）：

```
// CSS 的做法
.blast:nth-child(3) {
  color: red;
}
// jQuery 的做法
$(".blast").eq(2).css("color", "red");
```


上面两个例子都选中了页面上拥有.blast类的元素集合中，排在第三位的元素。（注意，jQuery的eq函数是从0开始计数的，而CSS中的nth-child是从1开始计数的，因此传入两者中的整数并不相同。）让我们继续采用jQuery函数来实现一个更复杂的例子：

```
<div>
  Current status: paused
</div>
// 以单词为分隔符拆分div
$("div").blast({ delimiter: "word" })
     // 选择句子中的第三个单词（也就是文本内容为paused的span元素）
     .eq(2)
       // 淡出第三个元素，然后用新信息替换里面的文本内容
       .velocity({ opacity: 0 }, function() { $(this).text("running"); })
       // 淡入替换后的文本
       .velocity({ opacity: 1 });
```

以上代码先将一个句子拆分，然后选择了第三个单词（paused），将其淡出，然后替换淡出

的单词为新单词（running），然后将新单词淡入。最终效果是句子中用于指示状态的关键词优美地替换成了新单词，提醒用户状态发生了改变。这是一个非常优雅的效果，但所用的代码仅几行而已。如果将这一效果多次用在一个较大的文本块上，得到的效果就是一条信息断断续续地变成了另一条信息。

5.5　华丽地过渡文本

迄今为止用到的都是transition.fadeIn过渡效果，其实你可以轻松地将其替换为Velocity UI pack中的其他效果。其中有一些还很华丽，包括transition.shrinkIn效果，它使元素由大变小进入视图；transition.perspectiveDownIn效果，它使元素像一扇百叶窗那样旋转向下进入视图。（永远要记住，不论你的效果多么复杂，都必须遵循第3章中讨论的那几条原则。）

注意　要获得UI pack所有效果的列表，包括在线示例，请访问VelocityJS.org/#uiPack。

请记住，有些效果使用了3D变换（rotateX、rotateY和translateZ），这些效果不能用于CSS中display值为"inline"的元素，尤其要注意span和anchor元素，因为它们display的默认值就是"inline"。有一种迂回解决的办法：将Blast生成的文本元素的display值设置为"inline-block"。这样做可以保持"inline"元素的正常行为，同时又额外给它们"block"元素（如div和p）特有的功能，可以为它们设置与位置相关的属性，包括3D变换在内。考虑到此display扭曲，进入视图的文本过渡示例现在就可以写成下面这样：

```
$("div")
  .html(message)
  .blast({ delimiter: "word" })
    .css({ opacity: 0, display: "inline-block" })
    .velocity("transition.perspectiveDownIn", { stagger: 50 });
```

这样就在jQuery的CSS()函数将元素opacity值设为初始值0的同一个调用中，将拆分后文本部分的display值设为了"inline-block"。

5.6　文字装饰

针对文本动画讨论的最后一个话题是装饰这一概念，就是在周围持续呈现效果，用以实现审美目的的动画。一个例子可能是一串文本像快要灭掉的灯泡一样一闪一闪。另一个例子可能是一

句话中所有单词的颜色总是从一种蓝变成另一种蓝。

这两种做法都是坏主意。

这种效果分散了用户的注意力，最终取悦的只是那些喜欢摆弄动效设计的开发人员自己。永远不要为了动画而动画；如果你的页面的某一部分毫无意义地将用户的注意力从实用功能那里吸引开，那就从头再来吧。

唯一的例外是状态指示器（比如“正在加载……”这样的文字），它可以让用户及时了解界面在做些什么。状态指示器适合进行文字装饰，因为这样的装饰会告诉用户界面正在处理数据（而不是卡在了那里）。通过这种方式，文字装饰就好像网页的心电图一样吸引人们的眼球。

既然文字装饰通常被认为是糟糕的做法，那么为什么还要把本节写在书里面呢？这是因为未使用动画效果的装饰经常是很棒的想法！把这当作Blast提供的非动画红利吧：你可以为Blast生成的文本元素添加样式，从而实现五彩拼贴或其他独特的排版设计。例如，可以把网站的标语文字（“将快乐送货上门！”）按单词拆开，然后依次减少每个单词的opacity值，这样就使整个句子产生了一种微妙的渐变效果。代码如下：

```
<div>
  Hi Mom
</div>
// 将div用Blast拆开，然后遍历生成的文本元素
$("div").blast({ delimiter: "character" }).each(function(i, element) {
  // 使用任意公式依次减少每个元素的oapcity值
  var adjustedOpacity = 1 - i/10;
  element.style.opacity = adjustedOpacity;
});
```

如果不迭代改变opacity值的话，也可以通过循环改变RGB值来创造一种基于颜色的渐变效果。例如，如果给原本灰色的文本逐渐增加蓝色分量，就会得到这样一组元素：从开头到结尾，蓝色变得越来越浓：

```
// 将div用Blast拆开，然后遍历生成的文本元素
$("div").blast({ delimiter: "character" }).each(function(i, element) {
  // 使用任意公式依次为每个元素增加蓝色分量
  var adjustedBlue = i*20;
  element.style.opacity = "rgb(0, 0," + adjustedBlue + ")";
});
```

记住：形式要服从功能。

5.7　小结

本章只是抛砖引玉地探讨了一下颗粒化文本控制的可能性。其他还有许多技巧值得探索，例如精细调节一个单词中每个字母的位置来生成拼贴效果，或者让单词围成一个圆圈来模仿你可能在杯垫上见到过的排版设计。

尽管这些技巧也许适合用到主页中央的大标题上，但它们可能并不适用于UI中需要与用户反复交互的重要板块上。为什么？这是因为有样式的文本比没有样式的文本更难一眼就看懂。但是，如果能考虑到形式与功能之间的平衡性，那么你会做得很好。

第6章 SVG入门

要深入地为SVG（可缩放矢量图）写教程，很容易就能写成一本书了。有鉴于此，本章只是为这个话题起个头儿，目的在于让你掌握充分的知识，可以顺畅地为SVG元素添加动画，还能了解从哪里能够继续深入学习。

6.1　用代码创建图片

SVG元素是DOM元素的一种类型，它借用了我们已熟知的HTML元素的写法来定义任意图形。SVG元素与HTML元素的不同之处在于它们具备特有的标签、属性和行为，可以用来定义图形。换言之，SVG使你能够用代码创建图片。这一概念无比强大，原因在于这意味着你可以使用JavaScript和CSS，以编程的方式为这些图形添加样式和动画。另外，SVG还具备很多其他优点。

- SVG的压缩性异常的好。用SVG定义的图形比同样的PNG/JPEG图片要小，这可以极大地缩短网站加载时间。
- SVG图形可以缩放至任意分辨率并且不损失清晰度。与其他标准图片格式不同，它在所有设备上都边缘清晰，该是向移动设备屏幕上的那些模糊的图片说再见的时候了。
- 就像HTML元素一样，你可以为SVG元素指定一个事件句柄，响应用户的输入。这意味着页面上的图片也可以具有交互性。如果你乐意，可以把网站上所有的按钮都换成动态图形。
- 许多照片编辑应用（包括Adobe Photoshop、Sketch和Inkscape）都可以将设计作品导出成SVG格式，可以直接复制粘贴到HTML当中。因此，即使你自认不是个艺术家，也可以借助第三方应用来自己做设计。

简而言之，SVG是不可思议的图形解决方案。接下来让我们继续深入了解！

6.2　SVG 标记的写法

SVG元素是用父级<svg>容器进行定义的。为该容器元素指定宽高，这样就为SVG图形渲染定义了画布：

```
<svg version="1.1" width="500" height="500" xmlns="http://www.w3.org/2000/svg">
  <circle cx="100" cy="100" r="30" />
  <rect id="rect" x="100" y="100" width="200" height="200" />
</svg>
```

在<svg>里面，可以插入各种SVG图形元素。上面的例子中有一个circle（圆形）元素，后面跟着一个rect（长方形）元素。正如普通的HTML元素，SVG元素也接受height和width属性，此处这些属性是用于演示目的。但是，就像HTML一样，SVG样式属性的设置最好在CSS样式表中完成。还是像HTML一样，样式表类通过其id、class或标签类型以SVG元素为目标。

SVG与HTML在设置上的本质不同在于它们所接受的HTML属性和CSS属性的范围是不同的。SVG元素只接受几个标准CSS属性。此外，SVG还接受一组被称为表象属性（presentational attribute）的特殊属性，包括fill、x和y。（fill指定图形填充的颜色；x和y定义了元素左上角的坐标。）这些属性定义了元素如何在画布上进行视觉呈现。让我们以rect作为SVG元素的例子，依次了解一下这些属性：

```
<rect id="rect" x="100" y="100" width="200" height="200" />
```

此处，width和height属性跟你预想的作用一样，不再赘述。特有的x和y属性定义了长方形在画布中的坐标。这些值简单地指示了长方形相对于原点（x=0, y=0）的位置。这里与HTML不同，SVG的定位不是通过CSS的top、right、bottom、left、float或margin属性来定义的；它的定位逻辑完全是通过明确定义的坐标来决定的。换句话说，一个SVG元素的定位不会影响到它的兄弟元素；它不会在页面上把兄弟元素挤到一边，而是会重叠起来。

现在，让我们看一下circle元素。它的渲染是通过圆形的圆心坐标（cx和cy）以及半径长度（r）决定的：

```
<circle cx="100" cy="100" r="30" />
```

很简单，对吗？SVG元素的标记结构与HTML元素相同，因此本章中所有的代码示例看起来都挺眼熟。

注意，SVG元素还有许多其他类型，包括ellipse（椭圆）、line（直线）和text（文本）。要了解更多详情，请参见本章结尾。

6.3 SVG 样式设置

SVG元素接受各种特殊的样式属性，这些属性并不适用于HTML元素。例如，SVG的fill属性类似于CSS中的background-color，stroke属性类似于border-color，stroke-width类似于border-width。让我们看下面这个例子：

```
<svg version="1.1" width="500" height="500" xmlns="http://www.w3.org/2000/svg">
  <circle cx="100" cy="100" r="30" style="fill: blue" />
  <rect id="rect" x="100" y="100" width="200" height="200" style="fill: green; stroke: red;
stroke-width: 5px" />
</svg>
```

上例中，`circle`元素是使用纯蓝色填充的，`rect`元素是使用纯绿色填充的。另外，长方形有一个5像素宽的红色边框。

SVG特有的样式属性还有许多。目前，你只需要知道它们存在，这样，在尝试设置SVG元素上的CSS属性的动画时，你可以额外留心一下。

注意 请参考本章最后的“小结”，查看所有SVG样式属性的完整列表。

6.4 对 SVG 的支持

现成的对SVG元素动画的支持都不太完美：不论jQuery还是CSS，对SVG特有的样式属性和表象属性都支持得不完全。另外，CSS过渡根本不能在Internet Explorer 9上设置SVG元素的动画；Internet Explorer的任何版本都不支持使用CSS将transform（变换）动画应用于SVG元素。

要想获得全面的SVG动画支持，要么使用专门的SVG库，要么使用内置了支持SVG元素的动画库。有一个专门的SVG库值得关注，那就是Snap.svg。另外，本书中一直使用的JavaScript动画库Velocity.js也提供了对SVG元素动画的完整支持，这点也许已在你意料之中。

注意 可前往SnapSVG.io下载Snap.svg库。

6.5 SVG 动画

SVG元素也许永远也成不了UI中的主力，但是在原本通常使用静态图片的位置使用它，确实会为页面增添一些情趣。SVG的用例包括下面几个。

- 具有复杂动画序列的按钮，用户悬停鼠标或点击鼠标时会触发这些动画序列。
- 独特的加载状态图形，可以用来替代老掉牙的转圈指示器GIF图。
- 公司logo，在页面加载时，logo的各个部分可以一起以动画形式呈现。

本章稍后会详细探讨最后一个用例。

6.5.1 传入属性

使用Velocity时，设置SVG属性的动画的方式与设置标准CSS属性的动画的方式相同。将适当的属性名称以及想要的最终值传入Velocity的属性对象中：

```
// 设置SVG元素填充红色和使用黑色边框的动画
$svgElement.velocity({ fill: "#ff0000", stroke: "#000000" });
```

对比之下，下面的代码就起不了作用，这是因为SVG元素并不支持其中的CSS属性：

```
// 错误：这些属性并不适用于SVG元素
$svgElement.velocity({ borderSize: "5px", borderColor: "#000000" });
```

6.5.2 表象属性

本章前面谈到过的表象属性也可以按照预期设置动画：

```
// 设置长方形的x、y坐标的动画
$("rect").velocity({ x: 100, y: 100 });
// 设置圆的cs、cy坐标的动画
$("circle").velocity({ cx: 100, cy: 100 });
// 设置长方形尺寸的动画
$("rect").velocity({ width: 200, height: 200 });
// 设置圆的半径的动画
$("circ").velocity({ r: 100 });
```

所有你正在使用的Velocity功能（比如动画反转、UI pack效果、序列触发等等）同样也适用于SVG元素。

6.5.3 定位属性（positional attribute） VS 变换（transform）

你可能想知道用定位属性（x、cx、y和cy）来设置SVG元素的位置与用CSS变换（例如translateX、translateY）有什么不同。答案是浏览器支持性不同。IE浏览器（直到IE 11）都不支持在SVG元素上使用CSS变换。请看下面的例子：

```
// 只要SVG元素支持，x、y属性就起作用（IE8以上版本，Android 3以上版本）
$("rect").velocity({ x: 100, y: 100 });
// 然而，定位变换 （例如translateX和translateY）只在**除IE浏览器之外**的其他浏览器上起作用
$("rect").velocity({ translateX: 100, translateY: 100 });
```

注意 尽管众所周知，变换由于硬件加速而性能格外出众（请阅读第7章了解详情），但上述两种SVG动画实现方式速度是一样快的，因为SVG图形默认就是硬件加速的。

SVG能让你走出长方形的桎梏。

6.6 应用实例：logo 动画

高分辨率网站的logo可以在页面加载完成时让动画运行到位，这是SVG实施的一个理想应用。假设你想简单地复制一下万事达信用卡的logo，该logo包括两个相互重叠的不同颜色的圆。

如果要使用Velocity让这个logo的动画运行到位，可以先定义SVG画布，代码如下：

```
<svg version="1.1" width="500" height="500" xmlns="http://www.w3.org/2000/svg">
  <circle id="circleLeft" cx="100" cy="100" r="30" style="fill: red" />
  <circle id="circleRight" cx="100" cy="100" r="30"style="fill: orange" />
</svg>
```

以上代码创建了两个半径相同并部分重叠的圆。接下来，要设置这两个圆从其初始位置向外远离的动画，使它们在运行动画之后，仅有轻微的重叠：

```
// 将其中一个圆向左移
$("#circleLeft").velocity({ cx: "-=15px" }, { easing: "spring" });
// 将另一个圆向右移
$("#circleRight").velocity({ cx: "+=15px" }, { easing: "spring" });
```

在这里，左边的圆向左移动了15像素（用"-="运算符指示Velocity从当前圆的cx值减去15）；右边的圆向右移动了15像素。spring缓动为动画增添了一点花样，使两个圆好像互相推挤着弹到两边。

既然SVG元素可以监听基于鼠标的事件（点击、悬停等），那么就可以进一步优化示例，把SVG元素变成可以交互的。在jQuery和Velocity的帮助下，SVG动画版应用就像下面这样实现了：

```
$("svg").on("mouseover mouseout", function() {
  $("#circleLeft, #circleRight").velocity("reverse");
});
```

以上代码在用户鼠标悬停在SVG元素上方或从其移开时触发了两个圆的页面加载动画的反转。利用Velocity的reverse动画命令，仅需一行代码就完成了这项工作。要了解reverse的更多详情，请参考第2章。事实上，当用户第一次鼠标移入时，进行的是与页面加载时相反的动画。当用户后来移出鼠标时，动画又再次反转了一次，使得logo恢复到了原来的形态。

尽管这个代码实例确实有点虎头蛇尾，不过它确实是个*好例子*，因为它反映了设置SVG元素的动画与设置HTML元素的动画之间的相似之处。SVG变得格外复杂的地方是想要定义任意形状而非正方形、长方形、圆形这种基本形状的时候。毕竟，SVG元素可以定义你在图片编辑器中可以想象到的任意图形，因此它们必须具有强大的表现力。而且它们也确实如此。但是，精通SVG设计已经超出了本书的范围。请看本章的“小结”，了解在哪里可以继续深入学习。

6.7 小结

如果你对读到的东西迷住了，想要学习有关SVG的更多知识，请看看以下优秀的资源。

- 要对SVG元素有个全面的认识，请参考Joni Trythall精彩绝伦并且免费的SVG口袋指南（https://github.com/jonitrythall/svgpocketguide）。
- 要想了解SVG元素所有的类型及属性，请访问Mozilla Developer Network（https://developer.mozilla.org/en-US/docs/Web/SVG）。
- 要想获取Velocity可以设置动画的所有SVG属性及样式属性列表，请参考VelocityJS.org/#svg。

第7章 动画性能

性能影响一切。性能提高，无论是表面上的还是实际上的，都能巨幅提升用户体验，继而提高公司的**赢利**。许多重要研究表明，搜索引擎的等待时间增长会显著降低每用户收入。直白点说，**人们厌恶等待**。

正如第1章中介绍的那样，JavaScript动画的性能并不亚于CSS动画。因此，如果使用了现代的动画库，例如Velocity，那么动画引擎的性能将不再是app的瓶颈，构成瓶颈的只有代码。而这正是本章所要探讨的：为所有浏览器和设备编写高性能动画代码的技巧。

7.1　网络性能的实际情况

如果想知道为什么同时运行动画会减慢UI速度，或者为什么你的网站在移动设备上变得很慢，你应该读读本章内容。

动画是浏览器运行中资源非常密集的进程，但是有很多技术能够帮助浏览器尽可能高效地运行。我们马上就会学到这些技术。

从UI设计的角度讲，盛赞移动优先、响应式网站的文章并没有任何不足。相反，从UI性能的角度讲，作为开发人员，我们中的大多数人都对提升性能的最佳实践和做法一无所知。想了解网络性能的最新现状需要了解铺天盖地的信息，而且通常都是徒劳；我们都已受缚于浏览器和设备的各种怪异模式，充斥在整个生态系统当中的各种设备（台式电脑、智能手机和平板电脑）和浏览器（Chrome、Android、Firefox、Safari和IE）的副产品。鉴于这些平台总是在更新，所以不出意料的是我们经常会举双手投降，把对于性能的顾虑尽可能放在一边。有时候，我们甚至可能会忍不住把所有动画一并去掉，因为拿不准它们会不会影响性能。

我们告诉自己：

既然设备在越变越快，而且用户也在不断更新他们的硬件，所以我的网站性能会逐渐提升。

但不幸的是，整个情况实际上是完全颠倒过来的：发展中国家所采用的智能手机，性能敌不过我们口袋里最新版的iPhone。难倒你真的想放弃为未来几十亿新网民创造产品吗？即将到来的Firefox OS已经摆开了架势，要为数百万人带来高性能的智能手机，因此我们不只是在美滋滋地想当然。移动革命就在当下。

注意　爱立信的报告称，全球的智能手机用户将在未来5年内从19亿增长到59亿，而其中绝大多数来自发展中国家。

性能
影响一切。

如果你的第一反应是“这不关我事，我的应用只是为发达国家中技术高超的中产阶级设计的”，那么你大可以放心，因为你那邪恶的网络开发同仁正坐在千里之遥不断盘算着，要付出必要努力在低端设备上提供优秀体验，从而先于你进入那个新兴市场。（事实上，致力于此的是一个庞大的开发军团，请在Google上搜索Rocket Internet。）

另外，忽视性能问题还会带来另一个堪忧的事实。那就是我们经常犯这样一个错误：总是在设备处于理想负载的情况下测试我们的网站。但事实上，毫无疑问，用户会同时开很多应用和浏览器标签页。他们的设备在任何指定时间都是在超时处理十几个任务。相应地，你为应用所设置的性能基准线并不符合真实世界的性能状况。哎呀！

但是别害怕，聪明的开发人员。是时候探索可用的性能技术，给你的动画编程升升级了。

7.2　技术：去除布局颠簸

布局颠簸，即DOM操作缺乏同步性，是拖垮动画性能的巨无霸。对它虽没有轻松的解决办法，但却有最佳实践。让我们继续来看。

7.2.1　问题

考虑一下网页操作是如何进行*设置*（setting）和*获取*（getting）这两项任务的：你可以*设置*（更新）或*获取*（查询）一个元素的CSS属性。同理，可以往页面里插入新元素（设置）或者从页面里查询一组已存在元素（获取）。获取和设置是引发性能开销的两个核心浏览器进程（另外还有图形渲染）。可以这样来想这个问题：在为元素设置了新属性以后，浏览器必须计算你这次更改所产生的后续影响。例如，改变一个元素的宽度会导致一系列连锁反应；它的父级元素、兄弟元素和子元素的宽度根据各自的CSS属性也要调整。

由设置和获取的交替而导致的UI性能降低被称为*布局颠簸*。尽管浏览器已经为页面布局的重新计算进行了高度优化，但由于布局颠簸，这些优化的效果大打折扣。例如，浏览器可以轻易地将同一时间的一系列获取操作优化成一个单一的、流畅的操作，这是因为浏览器在第一次获取之后可以缓存页面的状态，然后在后续每次获取操作时，参考那个状态。但是，如果反复执行了获取之后又执行设置，就会让浏览器去做许多繁重的工作，因为设置所做的更改会不断地使其缓存失效。

当布局颠簸在动画循环中出现的时候，对性能的影响更是雪上加霜。试想一个动画循环力求达到60帧每秒，这是人眼感知平滑运动的最低值。这意味着在动画循环中，每一个tick都必须在16.7毫秒（1秒/60tick≈16.67毫秒）内完成。布局颠簸很容易导致每个tick超过这个时限。最终结果当然就是动画变得磕磕巴巴。（用网页动画术语来讲就是*卡顿*。）

尽管有些动画引擎，例如Velocity.js，在其动画循环中为减少布局颠簸进行了优化，但还要当心在你自己的循环中避免出现布局颠簸，例如在`setInterval()`或自调用的`setTimeout()`代码里面。

7.2.2　解决办法

避免布局颠簸的方法很简单，那就是把DOM的设置和获取的操作分别集合在一起。以下代码会导致布局颠簸：

```
// 糟糕的做法
var currentTop = $("element").css("top"); // 获取
$("element").style.top = currentTop + 1; // 设置
var currentLeft = $("element").css("left"); // 获取
$("element").style.left = currentLeft + 1; // 设置
```

如果重写上述代码，把查询放在一起，把设置放在一起，那么浏览器就可以打包相应的操作，从而减少代码造成的布局颠簸的影响：

```
var currentTop = $("element").css("top"); // 获取
var currentLeft = $("element").css("left"); // 获取
$("element").css("top", currentTop + 1); // 设置
$("element").css("left", currentLeft + 1); // 设置
```

以上所说明的问题经常会在生产代码中看到，尤其是当UI操作依赖于元素当前CSS属性值的时候。

比如你的目的是在单击按钮的时候，切换侧边菜单的可见性。要想达到这一效果，你可能会先检查侧边菜单的display属性是设置成"none"还是"block"，然后再相应地进行值的替换。检查display属性的过程构成一次“获取”；后续不论是将侧边菜单显示出来还是隐藏起来都构成了一次“设置”。

要想优化这种代码就必须在内存中保留一个变量，每当按钮点击时，这个变量跟着更新，然后在切换可见性之前，通过查询这个变量得知侧边菜单的当前状态。这样，“获取”的过程就完全省掉了，从而有助于减少设置和获取交替出现的可能性。另外，除了降低布局颠簸发生的可能性以外，UI现在还得益于减少了一次页面查询。记住：每次设置和获取对于浏览器操作来说都比较消耗性能；设置和获取次数越少，UI的速度就会越快。

许许多多的小改进最终会积累成相当可观的好处，而这正是本章的潜在主题：尽可能多地遵循性能最佳实践，就可以尽可能少地为了性能而妥协自己心中理想的动效设计目标，从而实现满意的页面。

7.2.3 jQuery元素对象

实例化jQuery元素对象（jQuery element object，JEO）是造成DOM获取操作的常见真凶。你可能纳闷什么是JEO，但是你肯定见过下面这样的代码片段：

```
$("#element").css("opacity", 1);
```

或者等效的原生JavaScript：

```
document.getElementById("element").style.opacity = 1;
```

在jQuery代码中，由$("#element")返回的值就是一个JEO，即一个包装了所查询的原生DOM元素的对象。JEO提供了所有你欢喜的jQuery功能，包括.css()、.animate()等。

在原生代码中，由getElementById("element")返回的值是一个原生的（未包装的）DOM元素。上面两种写法都要求浏览器搜索DOM树，找到想要的元素。这种操作，如果重复多次，就会影响页面的性能。

当未被缓存的元素在重复使用的代码片段中出现，例如在循环代码中，对性能的影响就更加恶劣了。看下面这个例子：

```
$elements.each(function(i, element) {
  $("body").append(element);
});
```

你可以看到$("body")作为一个JEO实例，反复在$.each()的每次循环中出现：除了把循环中的当前元素添加到DOM（这么做也有其性能问题）以外，你还反复强制执行了DOM查询。这种看似无害的一行代码的操作累积起来会非常快。

解决办法就是缓存结果，或者把返回的JEO存到变量里去，这样就避免了每次在元素上调用jQuery函数时反复执行DOM操作。因此，代码从原来的这个样子：

```
// 糟糕做法：未缓存JEO
$("#element").css("opacity", 1);
// …… 一些中间代码……
// 我们再次将JEO实例化
$("#element").css("opacity", 0);
```

在恰当优化之后，变成下面这个样子：

```
// 缓存jQuery元素对象，在变量前面加个前缀$用来表示这是个JEO
var $element = $("#element");
$element.css("opacity", 1);
// …… 一些中间代码 ……
// 我们复用了缓存的JEO，避免了一次DOM查询
$element.css("opacity” , 0);
```

现在可以在整个代码里复用$element，并且不再会因它造成重复的DOM查询了。

7.2.4 强制给值

动画引擎的传统做法是在动画的一开始查询一遍DOM来确定每个被设置动画的CSS属性的初始值是多少。Velocity通过一种称为“强制给值”的功能可以绕过这一页面查询事件。这也是避免布局颠簸的另一项技术。通过强制给值，可以明确地为动画设置初始值，从而彻底免去了一开始就对页面进行获取的操作。

强制给定的值作为第二项被传入一个数组中，而这个数组替代了原本动画属性映射中属性值的位置。数组中的第一项是你想要设置动画变动到的最终值。

看看下面两个动画示例，两者都是在页面加载完成之后就触发的：

```
// 设置translateX从初始值0变动至500像素的动画
$element.velocity({ translateX: [ 500, 0 ] });
// 设置opacity从初始值1变动至0的动画
$element.velocity({ opacity: [ 0, 1 ]);
```

在第一个例子中，你强制给了translateX一个初始值0，因为你知道（页面刚刚开始加载）元素还待进行平移。强制给定的值可以是已知的初始值，也可以是想要的初始值。另外，在第二个例子中，元素的当前opacity值为1，因为这是opacity的默认值，而且你还未对元素进行任何修改。简而言之，有了强制给值的功能，可以在已了解元素样式的情况下减少浏览器的工作负担。

注意 只在动画序列第一次使用时，强制给动画属性一个值，而不是在后续动画发生的时候强制给值，因为Velocity已经在内部做了缓存工作：

```
$element
  // 在此处可以选择强制给值
  .velocity({ translateX: [ 500, 0 ] })
  // 不要在这里强制给值；500已经内部缓存起来了
  .velocity({ translateX: 1000 });
```


强制给值的功能在高压的情况下能体现无穷的价值，例如在桌面浏览器上同时运动大量元素的时候，或者用于处理低端移动设备上每个页面交互都会产生明显延迟的时候。

然而，在大多数实际UI动画场景中，不需要进行强制给值的优化，这是因为每当你在CSS样

式表中修改元素的值时，都要更新一遍强制给定的值，从而导致代码不那么容易维护。

注意　请参考第8章，大概了解强制给值的应用。

7.3　批量添加 DOM

正如减少布局颠簸一样，批量添加DOM是另一种有助于避免与浏览器发生多余交互的性能提升技巧。

7.3.1　问题

关于“获取”和“设置”的话题还没说完呢！有一种常见的页面设置操作是在页面运行时插入新DOM元素。为页面添加新元素有很多用途，不过其中最流行的也许就是无限滚动了，它在用户向下滚动的时候，不断让新元素在页面底部以动画方式进入视图。

在前一节中已经知晓，每当有一个新元素添加进来，浏览器就必须针对所有受到影响的元素进行计算。这是一个相对较慢的过程。因此，当每秒要进行多次DOM插入时，页面的性能就会受到显著影响。幸运的是，当处理多个元素时，如果所有元素是同时插入的，那么浏览器可以对这个设置的操作进行优化。但不幸的是，作为开发人员的我们经常无意识地放弃了这种优化做法，给DOM单独添加元素。请看下面未优化的DOM插入做法：

```
// 糟糕的做法
var $body = $("body");
var $newElements = [ "<div>Div 1</div>", "<div>Div 2</div>", "<div>Div 3</div>" ];
$newElements.each(function(i, element) {
  $(element).appendTo($body);
  // 其他代码
});
```

以上代码遍历了一组元素字符串，这组元素字符串被实例化到jQuery元素对象中。（这么做没有什么性能损失，因为你没有针对每个JEO去查询DOM。）然后，使用jQuery的appendTo()函数将每个元素插入到页面中。

问题是这样的：即使在appendTo()语句后面还有其他代码，浏览器也不会把这些DOM设置操作压缩成一个单一的插入操作，因为浏览器不能确定循环以外的异步代码操作不会在插入操作之

间修改DOM状态。例如，想象这样一个场景：在每次插入之后都查询DOM，想要搞清楚究竟有多少元素在页面上存在：

```
// 糟糕的做法
$newElements.each(function(i, element) {
  $(element).appendTo($body);
  // 输出body元素有多少个子元素
  console.log($body.children().size());
});
```

浏览器无法将上面的DOM插入优化成一次操作，这是因为代码明确要求浏览器告诉我们，在下次循环开始之前，究竟存在多少元素。因为浏览器每次都要返回正确数值，因此它无法批量处理后面所有的插入操作。

总之，在循环内部进行DOM元素插入时，每一次插入的操作与其他都是互相独立的，因此会造成明显的性能损失。

7.3.2　解决办法

不要将一个个新元素单独插入DOM中，而是要在内存中构建一个完整的DOM元素集合，然后把这个集合通过一个appendTo()调用插入到页面中。本节前面举例的代码经过优化后变成下面这样：

```
// 优化后
var $body = $("body");
var $newElements = [ "<div>Div 1</div>", "<div>Div 2</div>", "<div>Div 3</div>" ];
var html = "";
$newElements.each(function(i, element) {
  html += element;
});
$(html).appendTo($body);
```

以上代码将代表每个HTML元素的字符串连在一起形成一个主字符串，然后把这个主字符串转成JEO并一次性添加到DOM上。通过这种做法，浏览器得到明确指示，将所有元素一次性插入，相应的性能也得到了优化。

很简单，对吗？本章剩余的内容也是如此。提升性能的最佳实践通常都像这样简单。只需要训练好自己的眼睛，发现可以使用最佳实践的地方即可。

能否
提升性能
完全取决于
你自己。

7.4 技巧：避免影响临近的元素

提升性能很重要的一点就是要考虑一个元素的动画对其临近元素的影响。

7.4.1 问题

设置一个元素的尺寸的动画时，这种改变经常会影响附近元素的定位。例如，如果夹在两个兄弟元素之间的一个元素宽度缩小，那么它的兄弟元素的绝对定位就会动态改变，从而保持在动画元素的旁边。另一个例子可能是设置嵌入在父元素中的子元素的动画，而这个父元素并没有明确定义的width和height属性。相应地，设置子元素的动画时，父元素的尺寸也会改变，从而确保将子元素完全包裹在内。实际上，子元素并不是唯一被设置动画的元素，因为它的父元素的尺寸也被设置了动画。如果这发生在动画循环里面，那么浏览器在每次循环时要做的工作就更多了！

有很多CSS属性，一经改变，就会造成临近元素尺寸或位置的调整，其中包括：top、right、bottom和left，所有的margin和padding属性，border厚度，以及width和height尺寸。

作为关心性能的开发人员，你需要了解设置这些属性的动画会给页面带来什么影响。时刻问问自己，设置每个属性的动画会怎样影响临近元素。如果重写代码能够让你避免元素变化带来的互相影响，那么请考虑重写。事实上，要这么做有一种简便方法，继续看后面的解决办法！

7.4.2 解决办法

这种可以避免影响到临近元素的解决办法是尽可能设置CSS的transform属性（translateX、translateY、scaleX、scaleY、rotateZ、rotateX和rotateY）的动画。transform属性的特殊之处在于它们将目标元素提升至一个单独的层，这个层可以独立于页面其他内容单独渲染（通过GPU加速提升性能），因此相邻的元素不会受到影响。例如，在设置一个元素的translateX变动到"500px"的动画时，元素会向右移动500像素，覆盖在任何动画路径上已经存在的元素的上面。如果在动画路径上没有任何元素（也就是没有相邻的元素），那么使用translateX的效果与设置更慢的left属性的动画的效果，在页面上看起来是一样的。

因此，只要有可能，如果原本动画写成这样：

```
// 将元素自左侧移动500像素
$element.velocity({ left: "500px" });
```

就应该重构为以下代码：

```
// 更快：使用translateX
$element.velocity({ translateX: "500px" });
```

同理，如果你可以用translateY替代top，也请这样做：

```
$element.velocity({ top: "100px" });
// 更快：使用translateY
$element.velocity({ translateY: "100px" });
```


注意 有时候，你是有意使用left或top属性，以便可以相应改变相邻元素的位置。但在所有其他情况下，养成使用transform属性的习惯。对效率的提升作用会很明显。

优先考虑opacity胜于color

opacity是另一个可以让GPU渲染加速的CSS属性，因为它不影响元素的位置。因此，如果页面上的有些元素已经加了动画，比如，当用户鼠标悬停在元素上时color属性会改变，那么请考虑用设置opacity的动画来替代。如果最终效果跟设置颜色的动画的效果差不多优秀，那么考虑留用设置opacity的动画，因为这样做可以提升UI的性能，而不必妥协视觉效果。

作为关心性能的开发人员，你不能再随意选择动画属性了。你现在必须考虑选择每个属性所带来的影响。

注意　请查看CSSTrigger.com上的内容，了解每个CSS属性如何影响浏览器性能。

7.5　技巧：减少并发加载

浏览器有瓶颈。找到究竟在哪儿，尽量呆在瓶颈下边。

7.5.1　问题

当页面首次加载时，浏览器会尽可能快地处理HTML、CSS、JavaScript和图片。因此不出意外，这时候发生的动画容易发生延迟，它们在努力抢夺浏览器有限的资源。所以，尽管在页面加载序列中添加动画是显摆动效设计技巧的好时机，但如果想避免用户产生网站很慢的第一印象，那么克制自己不要这么做。

同理，当许多动画同时在页面上发生时，也会出现一个类似的并发性瓶颈，不论它是出现在页面生命周期中的哪个阶段。在这些情况下，浏览器在同时处理众多样式变化的重压下会喘不过气来，然后卡顿就发生了。

幸运的是，有一些聪明的技巧可以减少并发的动画加载。

7.5.2　解决办法

要解决并发问题有两种方法：错开动画以及将动画拆开放到一个序列里。

错开动画

减少并发动画加载的一个方式是使用Velocity的UI pack中的stagger功能，它会相继在一组元

素的动画开始前添加指定的延迟时间。例如，要设置一组元素中每个元素的opacity值变动至1的动画，并且在动画开始时间之间相继添加300毫秒的延迟，代码可能会是这样：

```
$elements.velocity({ opacity: 1 }, { stagger: 300 });
```

这时候，这些元素不再是完全同步执行动画的，而是在整个动画序列的开头，只有第一个元素在执行动画。然后，在整个序列的结尾，只有最后一个元素执行动画。你很高效地分散了动画序列的总工作量，使浏览器总是在每一刻做更少的工作，而不是同时执行每个元素的动画，让浏览器累得喘不过气来。另外，在动效设计中使用错开动画，通常会得到较好的审美效果。（请参考第3章，进一步了解错开动画的好处。）

多动画序列

减少并发加载还有另一个聪明的方法：将多个属性的动画拆成多动画序列。以设置元素的opacity值的动画为例。这通常是一个相对轻松的操作。但是，如果同时还要设置元素的width和box-shadow属性的动画，那么就会给浏览器带来更多可观的工作量：会影响更多像素，也要进行更多计算。

因此，如果原本动画像这样子：

```
$images.velocity({ opacity: 1, boxShadowBlur: "50px" });
```

也许可以考虑重构成下面这样：

```
$images
  .velocity({ opacity: 1 })
  .velocity({ boxShadowBlur: "50px" });
```

这样浏览器就有更少的并发工作要做，因为这些都是一个接一个发生的单独属性动画。注意此处要进行权衡，因为我们增长了整个动画序列的持续时间。这对于最终的应用场景而言，也许是好事，也许是坏事。

既然这种优化需要改变你原本对动效设计的想法，那么这一技巧并非总是要使用。把它作为最后的手段吧。如果需要在低端设备上挤出额外的性能，那么用这种技巧或许合适。其他情况下，不要用这种技巧预先优化网站上的代码，否则的话，最终得到的将是不必要的臃肿且晦涩的代码。

7.6　技巧：不用持续响应滚动（scroll）和调整大小（resize）事件

要注意代码运行的频率。如果一段很快的代码片段在每秒钟内运行1000次，那么累积起来，它也不会很快了。

7.6.1　问题

浏览器的滚动（scroll）和调整大小（resize）是两个触发频率非常频繁的事件类型：每当用户调整或滚动浏览器窗口时，浏览器都会在每秒内触发多次与这些事件相关的回调函数。因此，如果你注册的回调函数与DOM有交互的话，或者更糟，包含布局颠簸的话，那么它们会在滚动或调整大小时带来巨大的浏览器负担。请看下面的代码：

```
// 当滚动浏览器窗口时，执行一个行为
$(window).scroll(function() {
  // 这里写的任何行为都会在用户滚动时，每秒钟触发多次
});
// 当浏览器窗口的大小改变时，执行一个行为
$(window).resize(function() {
  // 这里写的任何行为都会在用户调整窗口大小时，每秒钟触发多次
});
```

要意识到，以上函数并不是在事件发生时只调用一次，而是在用户与页面进行相应交互的整段时间中，持续调用。

7.6.2　解决办法

这个问题的解决办法是对事件句柄进行*反跳*（debounce）。所谓反跳是指这样一个过程：定义一个时间间隔，在此时间间隔期间，事件句柄回调将仅会被调用一次。例如，假设你定义了一个250毫秒的反跳间隔，而用户滚动页面的总持续时间为1000毫秒。这时候，进行了反跳的事件句柄代码就会相应地仅触发四次（1000毫秒/250毫秒）。

反跳实施的代码已经超出了本书的范围。幸运的是，许多库专门用于解决这个问题。请访问davidwalsh.name/javascript-debounce-function查看例子。另外，非常流行的Underscore.js（UnderscoreJS.org），它是一个与jQuery很相近、也提供用于简化编程的辅助函数的JavaScript库，这个库含有debounce函数，你可以轻松地在事件句柄上反复使用它。

注意　就在编写本书期间，Chrome的最新版本已经自动反跳滚动事件了。

7.7　技巧：减少图片渲染

不是对所有元素的渲染都是一样的。要显示有些元素，浏览器要加班工作才行。让我们看看这些元素有哪些。

7.7.1　问题

视频和图片是多媒体元素类型，浏览器必须要加倍努力渲染才行。要计算非多媒体元素的尺寸属性是很轻松的，但是多媒体元素包含成千上万的像素数据，要改变它们的大小、尺寸或是重新合成，对浏览器而言计算开销是很大的。设置这些元素的动画的性能总是比不上设置标准HTML元素（如div、p和table）的动画的性能，来得理想。

另外，鉴于滚动页面几乎可以视为设置整个页面的动画（可以把滚动页面视为设置页面的top属性的动画），在CPU吃紧的移动设备上，多媒体元素也会造成滚动性能的巨幅下降。

7.7.2　解决办法

不幸的是，除了尽可能把简单的、基于图形的图片转成SVG元素以外，就没有其他任何办法可以将多媒体内容重构成更快的元素类型。因此，唯一可行的性能优化做法就是减少在页面上同时显示和同时设置动画的多媒体元素总数。注意这里用到的同时一词是在强调浏览器渲染的客观情况：浏览器只渲染可以看到的东西。页面上看不到的部分（包括包含额外图片的部分）是不会被渲染的，而且也不会对浏览器进程造成额外压力。

因此，有两种最佳实践可以遵循：第一种，如果原本感觉在页面上添不添额外图片都无所谓的话，那么选择不添。要渲染的图片越少，UI性能就越好。（更不用说更少的图片给页面网络加载时间带来的正面影响。）

第二种，如果你的UI在同时加载很多图片进入视图（比如，8幅或以上，根据设备硬件性能而定），考虑不要设置这些图片的动画，或者只是简单地切换每幅图片的可见性从不可见到可见。这种视觉效果可能并不优雅，要弥补这一点，可以考虑错开切换可见性的动画时间，使图片一个接一个显示而不是同时显示出来，这样做通常会产生出更精致的动效设计。

注意 请参考第3章，了解更多关于动画设计最佳实践的知识。

7.7.3 暗中潜入的图片

还没完呢。本节还有新东西要讲，因为我们还没探索完图片在页面上的呈现形式呢。最明显的性能杀手是img元素，但是图片还有另外两种方式可以暗中潜入到页面上来。

CSS渐变（gradient）

渐变实际上是图片的一种。它们不是用图片编辑器事先生成的，而是根据CSS的样式定义，在运行时生成的，例如在一个元素的`background-image`属性上用了`linear-gradient()`作为值。这里的解决办法是尽量选择纯色而非渐变背景。浏览器可以轻松优化纯色色块的渲染，但是就像对待图片一样，浏览器渲染渐变也格外费力，因为渐变的色彩是逐像素变化的。

阴影（shadow）属性

渐变有个邪恶的双胞胎，那就是`box-shadow`和`text-shadow`这两个CSS属性。它们的渲染跟渐变的渲染大同小异，只不过不是在`background-color`上，而是在`border-color`上罢了。更糟糕的是，它们的不透明度还逐渐减少，这要求浏览器进行额外的合成工作，因为渐变的半透明部分必须依据动画元素下面的元素来渲染。这里的解决办法跟之前的差不多：如果从样式表上移除这些CSS属性后，UI的视觉效果跟之前差不多优秀，那么宽慰一下自己，放弃之前的方案吧。网站的高性能会反过来回报你的。

这些建议只是建议而已。它们并非性能最佳实践，因为你要为了提高性能而牺牲设计本意。只有当网站性能很糟糕的时候，才考虑使用这些没有办法的办法。

7.8 在旧浏览器上降级动画

你不能直接忽略对性能低下的浏览器和设备的支持。如果从一开始就依照关注性能的工作流行事，你可以很简单地为它们提供降级体验，但保持功能齐全。

7.8.1 问题

IE8被称为一个龟速、过时的浏览器，正在逐渐失去人气。但它的后继者IE9还在美国以外被

广泛使用。另外，运行着Android 2.3.x及更早系统的老安卓智能手机比最新一代的Android和iOS设备要慢，但它们依然被广泛使用。网站的每十位用户中，估计有三位就属于IE9和老系统这两个阵营（主要依赖于你的应用所吸引的用户类型）。相应地，如果你的网站有丰富的动画和其他UI互动，那么就可以推断对于三分之一的用户来说，网站的运行很糟糕。

7.8.2 解决办法

要解决低端设备造成的性能问题有两种方式：要么不管三七二十一减少整个网站的动画；要么只针对低端设备减少动画。前者说到底是一种产品决策，而后者则只是一种可以轻松实施的技术决策，只要你使用了在第4章中介绍的全局动画乘数技术（或Velocity中对应的mock功能）。全局乘数技术使你能够通过一个变量改变整个网站的动画时间。因此，这里的诀窍就是：每当检测出用户正在使用性能较弱的浏览器，那么就将乘数设置为0（或者将`$.Velocity.mock`设置为`true`）。这样做就能让整个页面的动画都在一个动画tick（少于16毫秒）中完成：

```
// 使所有动画立即完成
$.Velocity.mock = true;
```

这种技术带来的结果就是将UI动画针对较弱的设备进行了降级，使原本的动态渐变换成了样式的立即更改。这么做的好处很明显：在页面上没有了资源密集型的动画效果，UI的运行会明显流畅许多。尽管这一技术无疑是具有破坏性的（因为它牺牲了你原本的动效设计意图），但是实用性方面的提升永远都值得用优雅性的降低来换取。毕竟，用户访问你的应用是为了完成某个目的，而不是为了欣赏UI效果多巧妙。永远不要让动画成为用户实现目的的绊脚石。

如果你仍然厌恶把动画从UI中剔除的想法，那么请牢记：使用较低端设备的用户已经习惯了网站慢速运行。因此，如果你的网站能够反其道而行之并取得积极效果，那么这些用户可能会感到格外欣喜，也将更有可能继续使用该网站。

7.9 尽早找到你的性能门限

这里要延续上一技术的主题，值得强调的是本章中的建议与移动设备特别相关，因为很多移动设备相对于台式电脑而言要更慢一些。但不幸的是，我们作为开发人员来说，经常在工作流中完全忽视这一点：我们已经习惯了在自己那运行着最新一代软硬件的高端台式机上用纯净的操作环境来创建网站。这种环境与真实世界中用户的环境是脱节的，因为他们通常使用的是过时的硬件和软件，而且倾向于在浏览器中同时运行多个标签页。换言之，我们大多数人的开发环境都有

很高的性能，无法代表实际用户的情况！这产生的副作用就是你根本意识不到自己的应用对于很可观的一部分用户来说是明显很慢的。等你去问他们在使用中遇到了什么挫折时，他们可能已经对这个网站完全失去兴趣了。

关心性能的开发人员会采取这样的正确方法：他会在开发初期就确定性能门限。当开发应用期间，频繁地用参考设备来进行测试，参考设备可能包括最新一代的移动设备，外加一个运行着IE9的虚拟机。如果早早地确定了你的应用在参考设备上要达到怎样的性能目标，之后你就可以放宽心了，因为你知道所有比参考设备更新的设备都能为用户提供更好的性能。

小窍门　如果你是Mac用户，请访问Microsoft的Modern.ie网站，了解有关如何运行虚拟的IE旧版本的信息。

如果你发现自己坚持想要的动效设计在参考设备上跑不起来，请遵循前面介绍的技巧：为那个参考设备将动画优雅降级，然后选择一个更快的设备（不需要降级动画的设备）作为新参考。

针对每一个测试设备，记住同时打开多个应用和标签页来模拟用户的操作环境。永远不要在真空中仅打开你自己的一个应用来测试。

请记住，远程浏览器测试（例如BrowserStack.com和SauceLabs.com所提供的服务）与真正的参考设备测试是不一样的。远程测试服务适合测试程序错误和UI的响应性，而不适合测试动画效果。毕竟，测试设备是在云端运行，并没有使用*真正的*硬件，它们只是模仿设备的不同版本罢了。因此，它们的性能与真正对应的设备性能通常是不同的。另外，在虚拟机上发生与在浏览器窗口中显示出来，这之间有明显延迟，因此你根本无法真正衡量UI动画的表现。

简而言之，为了性能测试，就真得跑出去买些真实设备回来。即使你口袋吃紧，也不能在这上面省钱。为了测试设备花的这几百美元会在今后用更多的经常性收入补回来，因为感到高兴的用户会更频繁地与你那顺滑的应用进行互动。

如果最终需要好几个参考设备，也可以考虑购买Device Lab支架，这个多功能支架能够支撑所有的移动设备，让它们在一个平面上显示，这样就可以轻松地在测试期间浏览这些屏幕了。作为福利，这个支架还有个漂亮的移动应用，它可以让你同时控制所有设备上的浏览器，这样就不用手动刷新每个浏览器的标签页了。

注意　请访问Vanamco.com购买和下载Device Lab。

尽早确定你的性能门限。

访问Ebay购买廉价老设备

购买最流行的Android和iOS设备主要发布周期的每一个产品，这会让你覆盖用户所拥有的各级别硬件和软件环境。以下是我推荐的测试装备（截至2015年初）。

- 运行iOS 7的iPhone 4或iPad 2
- 运行最新iOS版本的iPhone 5s或更新产品
- 运行Android 2.3.x的Motorola Droid X
- 运行Android 4.1.x的Samsung Galaxy SII
- 运行最新Android版本的Samsung Galaxy S5或更新产品

也可以将上述Android设备替换为任何相似性能的设备。这里的重点是你必须要使用涵盖每个重要Android发布周期的设备（2.3.x、4.1.x等），这样每个设备上的网络浏览器性能才具有代表性。请参考http://developer.android.com/about/dashboards，了解更多最流行的Android版本的分布。

7.10 小结

性能影响一切。从多少设备可以运行你的应用，到用户体验的质量，再到应用技术实力的感性认识，性能是专业网络设计的一个重要宗旨。它不仅仅是“有了更好”，而是根本的基石。不要把性能看扁了，而认为它只是事后进行的简单优化。

第8章 动画演示

是时候撸起袖子大干一场了！最后一章将带你实现一个由Velocity实现的完整动画演示。在阅读演示代码期间，你会学到如何综合运用Velocity的核心功能，大幅优化UI动画的工作流。这一演示也会为你介绍操作CSS中transform属性的高级技巧，这对当今以网页为基础的动画而言至关重要。

简而言之，你马上就要使用在本书中积累的技能去做一些酷炫的事情了。在写演示代码期间，有两个目标：一是使用简洁、表意的动画代码，二是确保最佳性能。

8.1 行为

演示中包含250个从屏幕飘进飘出、飘来飘去的圆形。过一段时间，画面会放大，虚拟镜头推进，然后虚拟镜头后撤至最初的位置。最初显示的画面会短暂显示放大的效果。

注意 在继续读下去之前，请前往VelocityJS.org/demo.book.html预览演示。（可以在页面上任意位置单击鼠标右键，然后选择“查看源代码”来查看演示的代码。）

圆形元素只是普通的`div`，在CSS中对这些`div`的`box-shadow`和`border-radius`进行了设置。此处没有用到WebGL或Canvas动画，只是单纯的HTML元素操作。（鉴于同时设置动画的元素数量很大，这个演示能够在DOM中运行得如此流畅是很让人印象深刻的。）

让我们将这个动画拆解开看：它包含了一些沿着*X*、*Y*和*Z*轴平移的`div`元素。*Z*轴确定了每个元素动画的深度；而*X*和*Y*轴则提供了在屏幕上看到的平滑的二维运动。与元素的各个运动同时发生的还有更宏观的一个视角转变，它发生在含有所有这些`div`的容器元素上。这一视角转变每3秒发生一次，从而营造了周期性的缩放效果，使观看者感觉好像正在圆形所处的三维空间中遨游。

第二个画面描绘的是缩小后的3D屏幕，与最初画面放大后的效果形成对比。

如何下载示例代码

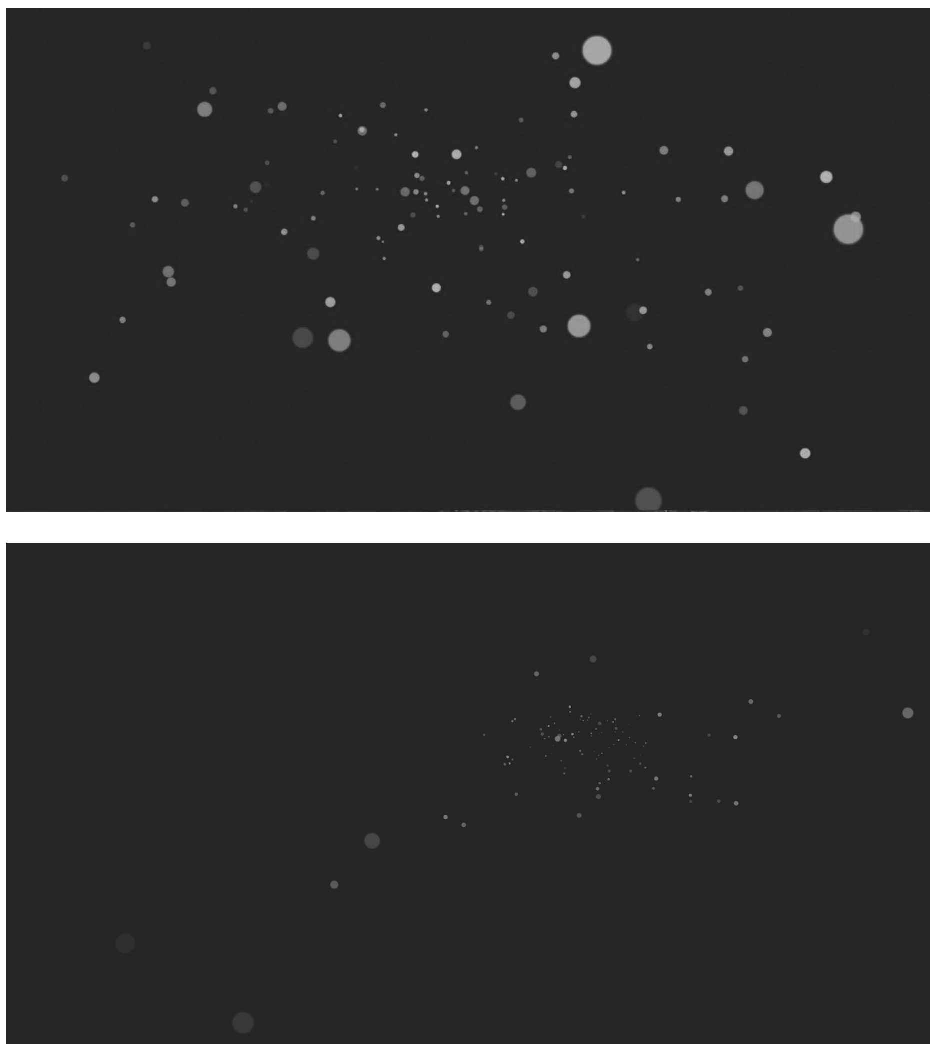
动画演示的代码可以从peachpit.com下载得到。以下是获取方式。

(1) 前往www.peachpit.com/register创建或登录账号。

(2) 输入图书的ISBN号（978-0-13-409670-4），然后单击Submit（提交）。

(3) My Registered Products（我的注册产品）页面打开。在列表中找到此书，然后单击Access Bonus Content（获取奖励内容）。

(4) 包含下载链接的页面打开，请单击获取名为WebAnimationJS_DemoCodeSample.zip的动画演示文件。



8.2 代码结构

让我们看一下实现该演示的代码。它的结构如下。

(1) 动画设置：用于约束动画移动的参数规范。

(2) 圆形创建：生成要设置动画的div元素。

(3) 容器动画：负责设置圆形的父元素的动画的代码。

(4) 圆形动画：负责设置圆形元素本身的动画的代码。

尝试熟悉该演示实施的宏观结构，这样在接下来几节中探究每个单独的代码段时，你的心中就有了全局的概念：

```
/************************
    动画设置
************************/
/* 在两个数之间随机生成一个整数。 */
function r (min, max) {
 return Math.floor(Math.random() * (max - min + 1)) + min;
}
/* 查询窗口的尺寸。 */
var screenWidth = window.screen.availWidth,
  screenHeight = window.screen.availHeight;
/* 定义Z轴动画范围。 */
var translateZMin = -725,
  translateZMax = 600;

  /**********************
    圆形创建
**********************/
var circleCount = 250,
  circlesHtml = "",
  $circles = "";
for (var i = 0; i < circleCount; i++) {
  circlesHtml += "<div class='circle'></div>";
}
$circle = $(circlesHtml);

/*****************************
      容器动画
*****************************/
$container
  .css("perspective-origin", screenWidth/2 + "px " +  screenHeight/2 + "px")
  .velocity(
  {
     perspective: [ 215, 50 ],
     opacity: [ 0.90, 0.55 ]
  }, {
     duration: 800,
     loop: 1,
     delay: 3000
  });

  /***************************
```

```
    圆形动画
***************************/
$circles
  .appendTo($container)
  .velocity({
   opacity: [
      function() { return Math.random() },
      function() { return Math.random() + 0.1 }
   ],
   translateX: [
      function() { return "+=" + r(-screenWidth/2.5, screenWidth/2.5) },
      function() { return r(0, screenWidth) }
   ],
   translateY: [
      function() { return "+=" + r(-screenHeight/2.75,  screenHeight/2.75) },
      function() { return r(0, screenHeight) }
   ],
   translateZ: [
      function() { return "+=" + r(translateZMin, translateZMax) },
      function() { return r(translateZMin, translateZMax) }
   ]
}, { duration: 6000 })
velocity("reverse", { easing: "easeOutQuad" })
velocity({ opacity: 0 }, 2000);
```

8.3 代码段：动画设置

为了参考方便，将这一部分的代码复制如下：

```
/************************
    动画设置
************************/
/* 在两个数之间随机生成一个整数。 */
function r (min, max) {
 return Math.floor(Math.random() * (max - min + 1)) + min;
}
/* 查询窗口的尺寸。*/
var screenWidth = window.screen.availWidth,
  screenHeight = window.screen.availHeight;
/* 定义Z轴动画范围。 */
var translateZMin = -725,
  translateZMax = 600;
```

第一部分“动画设置”在一开始定义了一个函数r（"random"的缩写），使用该函数可以人工限制随机生成的整数值的范围。该函数接受min和max两个参数，然后会输出一个在min到max之间的随机整数。（这里使用了一些基本代数知识。）当你在后面两个代码段中预设的范围内随机设定动画元素CSS的transform值时，会用到这个整数。

接下来的代码查询了window对象，获取了显示器的尺寸。通过稍后参考这些值，可以确保圆形的动画不会远到屏幕之外（并因此而看不到）。

动画设置的结果是定义了元素在Z轴上移动的min（最小值）和max（最大值）。这些值控制了你想要设置元素的动画从初始大小变动到多小（多么远离）或多大（多么靠近）。确切地说，这段代码规定圆形沿着Z轴可以最远远离虚拟镜头（或说远离屏幕）725像素，最近不得超过600像素。在这个案例中并没有限制圆形元素不得离开屏幕，但是圆形可能离得太远以至于看不到；或者离得太近以至于占据整个屏幕。从根本上说，这里由你自己的创意决定。

8.4　代码段：圆形创建

```
  /**********************
    圆形创建
**********************/
var circleCount = 250,
  circlesHtml = "",
  $circles = "";
for (var i = 0; i < circleCount; i++) {
  circlesHtml += "<div class='circle'></div>";
}
$circle = $(circlesHtml);
```

这是演示的第二部分代码：圆形创建。它生成了要设置动画的基本div元素。在这里，首先将理想的圆形数量定义为circleCount。然后，定义了circlesHtml字符串，用于容纳拼接之后的圆形的HTML代码。

然后，代码从0循环至circleCount的数字，生成圆形的HTML代码。注意，这里使用了第7章中介绍过的批量添加DOM的性能最佳实践。它将每个div元素的HTML代码拼接成一个主circlesHTML字符串，然后用一次操作将其插入到DOM中去。（如果一次只往DOM里添加一个元素，那么对性能产生的负面影响是很显著的：浏览器中的UI互动会僵住，直至相对较慢的元素插入过程完成。）

最后，代码将圆形元素包在一个jQuery元素对象里面，这样它们就可以作为一组元素在后续的圆形动画代码部分轻松操作了。

8.5 代码段：容器动画

```
/******************************
      容器动画
******************************/
$container
  .css("perspective-origin", screenWidth/2 + "px " +  screenHeight/2 + "px")
  .velocity(
  {
     perspective: [ 215, 50 ],
     opacity: [ 0.90, 0.55 ]
}, {
    duration: 800,
    loop: 1,
    delay: 3000
  });
```

8.5.1 三维CSS入门

让我们看看代码库中两段有关动画的代码中的第一段，这部分关注的是包含圆形元素的父级元素。在我们深入研究代码之前，先学习一下有关浏览器中三维动画的入门知识。

为了使transform在三维上发挥作用（例如：`translateZ`、`rotateX`、`rotateY`），必须在CSS中，为父级元素添加`perspective`属性。在此例中，`$container`元素正是派这个用场。

为`perspective`设置的值越大，*Z*轴平移（通过CSS的`translateZ`设置）看起来相对于其原点移动的距离就越短。换言之，如果想要在三维动画中强调深度，就将父元素的`perspective`属性设置为较小的数，比如50px，这实际上也是演示中为容器元素设置的值。相比之下，如果将`perspective`设为一个较大的值，例如250px，那么元素的`translateZ`每增加一个像素所产生的相对于原点的可见性移动就会变小。

另一个单独的补充性CSS属性就是`prospective-origin`，它定义了虚拟镜头放置的角度。虚拟镜头是一个窥视孔，观看者通过它观看浏览器中的三维动画。这一部分代码使用了jQuery的`$.css()`函数在容器元素上设置`perspective-origin`值，使虚拟镜头放在了页面的正中央，从而营

造了一种垂直于三维动画的视角。这个视角使用户看上去感觉圆形在正对着他的方向上飞近或飞远。

具体来说，该代码段将perspective-origin在页面上设置在浏览器宽度和高度一半的位置，即页面的中心点。这里用到了“动画设置”代码段中查询到的window尺寸。

了解了这些背景知识后，让我们来研究一下这段代码。

8.5.2 属性

这一部分代码创造了演示中放大或缩小的效果。为了参考方便，重新复制在下面：

```
$container
  .css("perspective-origin", screenWidth/2 + "px " +  screenHeight/2 + "px")
  .velocity(
  {
     perspective: [ 215, 50 ],
     opacity: [ 0.90, 0.55 ]
  }, {
     duration: 800,
     loop: 1,
     delay: 3250
  });
```

容器元素上的prospective-origin属性一设置完成，就用Velocity来设置其prospective属性的动画。这么做很有必要，因为想要得到的演示效果并不是在静止场景中保持优势位置（垂直视角），而是要先扩大后缩小元素与虚拟镜头的距离，这时候就是perspective属性发挥作用的时候了。

具体来说，这部分代码用Velocity将元素的perspective属性从初始值50px变动至最终值215px。

通过传入一个数组作为动画属性的值，可以强制指定设置属性toward的动画的最终值（上面的例子中是215px），还可以指定设置属性from的动画的初始值（上面的例子中是50px）。尽管完全可以只传入一个整数作为属性值，而且一般Velocity都是这么用的，但是“强制给值”的写法能够使你更好地控制属性的完整动画路径。

你可能有疑问，难倒“强制给值”不是多此一举吗？毕竟Velocity知道如何自动获取CSS属性的初始值。尽管在只传入一个值而不是一个数组时，Velocity默认会获取初始值，但这并不总是

理想的操作行为，因为向DOM查询属性的初始值具有潜在的性能问题。强制给值的写法允许你明确传入一个已知的初始值，从而避免Velocity去查询DOM。换言之，在perspective属性中设置的初始值50px与你一开始在样式表中为容器元素设置的perspective值一致。你只是在此重复了一下这个值而已。注意，在元素的opacity属性上也使用了强制给值的技巧：opacity是从初始值0.55变动至最终值0.90，而0.55正是在CSS中设置的opacity属性值。

正如在第7章中充分讨论过的，DOM查询是动画性能的致命弱点，因为浏览器要进行资源密集型运算来确定元素的视觉状态。尽管在这个演示中，Velocity动画不是在一个循环中被反复触发的，因此添不添加这种性能优化并不重要，但在这里还是把它加了进来，用于与强制给值的第二种应用作对比。本章后面的部分将会介绍强制给值的第二个用途。

设置perspective和opacity的动画的最终效果是容器的所有圆形元素看起来都在向虚拟镜头的方向拉近，同时执行亮度增加的动画（opacity从0.55变动至0.90）。不透明度的增加模拟了真实世界中光的行为方式：观看者离发光体越近，发光体看上去就越亮。

8.5.3 选项

容器动画代码的最后一部分是传入Velocity的选项：duration，这个不用再解释了；delay，它在动画开始之前插入一个暂停时间；loop，它在属性映射中定义的值与动画开始之前元素的值之间来回循环动画。具体来说，通过将loop设置为2，就是在告诉Velocity要变动至属性映射中设置的值，然后变动回之前的初始值，然后在3000毫秒的延迟之后再重复整个循环迭代一次。

注意 当delay与loop同时设置时，延迟会发生在每次循环之间。使用延迟可以创造一种愉悦的暂停感受，使缩放效果看上去不是突兀地来回切换。

8.6 代码段：圆形动画

在这里，事情变得有趣起来了。让我们看一下圆形动画，在这里要同时分别设置它们在*X*、*Y*和*Z*轴上平移的动画。你还将设置其opacity值的动画。

```
  /***************************
      圆形动画
***************************/
$circles
  .appendTo($container)
```

```
.velocity({
   opacity: [
     function() { return Math.random() },
     function() { return Math.random() + 0.1 }
   ],
   translateX: [
     function() { return "+=" + r(-screenWidth/2.5, screenWidth/2.5) },
     function() { return r(0, screenWidth) }
   ],
   translateY: [
     function() { return "+=" + r(-screenHeight/2.75,  screenHeight/2.75) },
     function() { return r(0, screenHeight) }
   ],
   translateZ: [
     function() { return "+=" + r(translateZMin, translateZMax) },
     function() { return r(translateZMin, translateZMax) }
   ]
}, { duration: 6000 })
.velocity("reverse", { easing: "easeOutQuad" })
.velocity({ opacity: 0 }, 2000);
```

8.6.1　值函数

与上一个代码段中用到的静态动画属性值（例如[215, 50]）不同，此代码段将函数用于属性值：会强制赋予每个属性一个数组，数组的初始值和最终值是由函数动态生成的。让我们简要了解一下值函数，这也是Velocity的一个独特功能。

注意　请访问VelocityJS.org/#valueFunctions了解更多有关值函数的内容。

值函数使你可以将函数传入为动画属性值。这些函数在运行时触发，并为集合中设置了动画的每个元素分别调用。在演示中，所讨论的集合就是$circles这个jQuery元素对象里所包含的圆形div。最终，当动画开始时，每个圆形元素的属性都会分别设置为随机值。另外还有唯一的一种方法可以使一个集合中每个元素的动画属性都有所不同，那就是通过循环分别设置每个元素的动画，但这种方法的代码乱七八糟，效果也很糟糕。这就体现了值函数的优越性，使用值函数能够保持动画代码简洁、易维护。

请注意，为了生成随机值，本部分代码利用了在“动画设置”代码段中定义的辅助函数r。（提醒一下，r函数返回的是min和max参数之间的随机整数。）你马上就会学习到有关这个函数的更多内容。

8.6.2 不透明度动画

不透明度属性从一个随机值变动到另一个随机值。在初始值这里，给随机值加上了0.1以确保元素不会透明得太厉害以至于看不出来了，因为毕竟你想让人看到你做的是什么元素的动画效果！opacity的变动导致分散在页面上的圆形从动画的第一帧开始就具有不同的不透明度。各不相同的不透明度营造了一种优美的渐变效果，为演示增添了丰富的视觉感受。

这段代码使用强制给值技巧实现了性能优化以外的另一目的：即强制给插入DOM的元素属性赋予了值函数，从而可以动态生成初始值。这意味着你成功避免了编写一段全新的代码，只为了给圆形元素设置CSS初始状态。你在负责设置独特初始位置的动画的同一行代码中动态地提供了这些位置。正如在第4章中充分讨论过的，应该努力让所有代码都像这样表达明确的意义。

8.6.3 平移动画

为了参考方便，此处重复一下这部分代码：

```
  /***************************
      圆形动画
***************************/
$circles
  .appendTo($container)
  .velocity({
     opacity: [
       function() { return Math.random() },
       function() { return Math.random() + 0.1 }
     ],
     translateX: [
       function() { return "+=" + r(-screenWidth/2.5, screenWidth/2.5) },
       function() { return r(0, screenWidth) }
     ],
     translateY: [
       function() { return "+=" + r(-screenHeight/2.75,  screenHeight/2.75) },
       function() { return r(0, screenHeight) }
     ],
     translateZ: [
       function() { return "+=" + r(translateZMin, translateZMax) },
       function() { return r(translateZMin, translateZMax) }
     ]
  }, { duration: 6000 })
  .velocity("reverse", { easing: "easeOutQuad" })
  .velocity({ opacity: 0 }, 2000);
```

是时候该仔细查看一下translate动画了，它在演示的三维空间中分别平移了圆形元素的位置。在所有三个轴的方向上，都是从一个随机初始值移动到一个随机最终值。这里的运算符包含一个加号后面跟一个等号（+=），它告诉动画引擎从初始值给动画属性增加一个单位。换言之，+=值运算符指示动画引擎将最终值视为一个相对值。相比之下，动画引擎的默认行为是从绝对意义上解释最终值。

就像opacity属性一样，这段代码也利用了强制给值和值函数，使代码不仅表意明确而且性能优越。尤其值得一提的是，圆形的移动在*X*和*Y*轴上被限制在与屏幕尺寸相关的范围内，在*Z*轴上，被限制在与最浅深度和最深深度相关的范围内。（提醒一下，最浅深度和最深深度是在“动画设置”那段代码中设置的。）在*X*和*Y*轴上用到一个随意的容差系数（请注意除数2.75）来减少元素执行动画的分散度。这个值是个创意决策，可以根据自己的审美偏好进行更改。

在代码最后，选项对象为整个动画设置了6000毫秒的持续时间。

8.6.4 反转命令

在主Velocity动画调用之后，链式操作继续调用了Velocity的反转命令。反转并不完全像你想象的那样：会设置目标元素回到上一个Velocity调用发生之前的初始值的动画。在这个特殊案例中，由于初始值是在上一个Velocity调用中强制给定的，因此那些初始值才是反转命令将要设置动画回到的初始值。

我选择在演示中包含反转命令的一个原因是要通过一行易于维护的表意代码延长整个动画的持续时间。（尽管你会质疑，将动画的持续时间从6000毫秒延长至12 000毫秒，这会减慢圆形移动的速度。）使用反转命令的方便之处在于它避免了重新设置（即手工输入）所有动画的初始值。重新手工设置初始值其实是个庞大、繁杂的工程，因为你首先要把所有随机生成的初始值都存到内存中，然后才能再设置变动至这些值的动画。因此，反转是Velocity的另一个很棒的功能，使演示用区区几行代码就可以完成很多工作。

Velocity的反转命令默认采用在上一个Velocity调用中使用的选项对象，包括duration和easing等。在这种情况下，由于上一个Velocity调用中使用的持续时间设置为6000ms，因此反转调用也将使用相同的设置。反转命令还允许你指定一个新的选项对象以扩展到上一个调用上。此演示使用了一种新的缓动类型easeOutQuad，在动画的反转方向上增添了新的动效设计能力。

小窍门 要预览各种流行缓动类型的行为，请访问http://easings.net。

当反转动画完成之后，一个最终的Velocity调用通过用2000毫秒时间将元素的`opacity`值过渡至0，使元素淡出视图。这使演示动画结束后的浏览器画布与开始时的视觉状态一致：一干二净、空空如也！你的工作也就到此结束了。

8.7 小结

从强制给值到值函数，再到反转，这个演示带你了解了Velocity动画引擎的强大功能。本书的重点放在了Velocity身上，希望本章过后你会确信这么做是值得的。使用不足75行的代码，就创造了不同于以往见到的纯HTML的丰富3D场景，而且代码不仅简洁、表意而且高效。

这个例子充分反映出：要实现看似复杂的动画其实很简单，尤其是当你使用了正确的工具并遵循了最佳实践。我的希望是将你在网络上看到的优美动画总结为一系列容易掌握并可以用在自己动效设计中的原则。

现在，去设计一些美丽的网站和应用吧！一旦搞出了什么酷炫的效果，别忘了在Twitter上展示给我看：twitter.com/shapiro。

要继续学习，
请关注
@shapiro。

延 展 阅 读

Web Animation using JavaScript

Develop and Design

JavaScript 网页动画设计

“我是一名资深的前端开发人员，阅读过的相关技术书已经记不清有多少本，但这本书仍让我眼前一亮，受益匪浅。目前市面上还没有其他书论述这个主题，更遑论对文本动画、动画工作流、动画性能等问题的深入讨论，凭这一点，本书也值五星！”

——David Credo，前端开发工程师

网络时代，用户体验的重要性毋庸置疑，动画在这一过程中的重要性也明显提升。如何在不分散用户注意的情况下达到动画设计加强页面目的的效果，已经成为优秀的用户界面设计师和Web开发人员孜孜以求的目标。本书将为此提供必备的知识。

书中内容共分为8章，以作者开发的动画库Velocity.js为工具，简明扼要地探讨了JavaScript动画的特点和工作流方面的优势，涵盖开发者们最关心的文本动画、SVG、动画性能等问题。掌握书中内容，即可自信实现视觉上效果震撼、技术上易于维护的动画效果。



图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机/程序设计/Web开发

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-41012-2
定价：39.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks